LINE: Queueing Analysis Algorithms

Example Manual – Python Edition

Last revision: September 10, 2026
QORE Lab, Imperial College London

Contents

1 About this manual **1**

2 Basic models **2**

2.1 Open queueing networks . . . 2
2.1.1 oqn_basic . . . 2
2.1.2 oqn_cs_routing . . . 4
2.1.3 oqn_fourqueues . . . 5
2.1.4 oqn_mapt . . . 6
2.1.5 oqn_multichain_cs . . . 7
2.1.6 oqn_nhpp . . . 8
2.1.7 oqn_oneline . . . 9
2.1.8 oqn_trace_driven . . . 10
2.1.9 oqn_vsinks . . . 11
2.2 Closed queueing networks . . . 13
2.2.1 cqn_bas_blocking . . . 13
2.2.2 cqn_bcmp_theorem . . . 14
2.2.3 cqn_lcfs_multiclass . . . 15
2.2.4 cqn_mmpp2_service . . . 17
2.2.5 cqn_multichain_cs . . . 18
2.2.6 cqn_multiserver . . . 20
2.2.7 cqn_multiserver_nc . . . 21
2.2.8 cqn_oneline . . . 23
2.2.9 cqn_repairmen . . . 24
2.2.10 cqn_repairmen_multi . . . 25
2.2.11 cqn_scheduling_dps . . . 26
2.2.12 cqn_threeclass_hyperl . . . 28
2.2.13 cqn_twoclass_erl . . . 29
2.2.14 cqn_twoclass_hyperl . . . 31
2.2.15 cqn_twoqueues_multi . . . 32
2.3 Mixed networks . . . 34
2.3.1 mqn_basic . . . 34
2.3.2 mqn_multichain_cs . . . 35
2.3.3 mqn_multiserver_fcfs . . . 37
2.3.4 mqn_multiserver_ps . . . 39
2.3.5 mqn_singleserver_fcfs . . . 41
2.3.6 mqn_singleserver_ps . . . 42
2.4 Class switching . . . 43
2.4.1 cs_implicit . . . 43
2.4.2 cs_multi_diamond . . . 44

2.4.3 cs_single_diamond . . . 46
2.4.4 cs_transient_class . . . 47
2.5 Priority models . . . 49
2.5.1 prio_hol_closed . . . 50
2.5.2 prio_hol_open . . . 51
2.5.3 prio_identical . . . 54
2.5.4 prio_psprio . . . 54
2.6 Fork-join networks . . . 56
2.6.1 fj_asymm . . . 56
2.6.2 fj_basic_closed . . . 57
2.6.3 fj_basic_nesting . . . 59
2.6.4 fj_basic_open . . . 61
2.6.5 fj_complex_serial . . . 62
2.6.6 fj_cs_multi_visits . . . 64
2.6.7 fj_cs_postfork . . . 65
2.6.8 fj_cs_prefork . . . 67
2.6.9 fj_deep_nesting . . . 68
2.6.10 fj_delays . . . 70
2.6.11 fj_mixed_openclosed . . . 71
2.6.12 fj_nojoin . . . 74
2.6.13 fj_route_overlap . . . 75
2.6.14 fj_serialfjs_closed . . . 77
2.6.15 fj_serialfjs_open . . . 78
2.6.16 fj_threebranches . . . 80
2.6.17 fj_tiny_closed . . . 82
2.6.18 fj_twoclasses_forked . . . 83
2.6.19 fj_variable_fanout . . . 85
2.7 Layered queueing networks . . . 86
2.7.1 lqn_basic . . . 86
2.7.2 lqn_bpmn . . . 87
2.7.3 lqn_bpmn_trace . . . 90
2.7.4 lqn_flatph . . . 92
2.7.5 lqn_fork_open_arrival . . . 93
2.7.6 lqn_jsq . . . 94
2.7.7 lqn_moment3 . . . 95
2.7.8 lqn_multi_solvers . . . 97
2.7.9 lqn_ofbiz . . . 98
2.7.10 lqn_open_arrival . . . 101
2.7.11 lqn_paramident . . . 102
2.7.12 lqn_rrobin . . . 102
2.7.13 lqn_serial . . . 103
2.7.14 lqn_server_pools . . . 104
2.7.15 lqn_setup . . . 105
2.7.16 lqn_sockshop . . . 107
2.7.17 lqn_srvnph . . . 109
2.7.18 lqn_transient . . . 109
2.7.19 lqn_twotasks . . . 110
2.7.20 lqn_workflows . . . 112
2.8 Cache models . . . 113
2.8.1 cache_compare_replc . . . 114

2.8.2 cache_itemsize_costcap . . . 115
2.8.3 cache_mmap_rr_env . . . 116
2.8.4 cache_replc_climb . . . 117
2.8.5 cache_replc_fifo . . . 118
2.8.6 cache_replc_hlru . . . 120
2.8.7 cache_replc_lru . . . 121
2.8.8 cache_replc_qlru . . . 123
2.8.9 cache_replc_routing . . . 124
2.8.10 cache_replc_rr . . . 125
2.8.11 cache_rmf_transient . . . 127
2.8.12 cachenet_tandem . . . 128
2.8.13 retrieval_chain . . . 130
2.8.14 retrieval_default . . . 131
2.8.15 retrieval_ps . . . 134
2.8.16 retrieval_routing . . . 135
2.8.17 retrieval_simple . . . 138
2.9 Cluster models . . . 139
2.9.1 cl_basic . . . 139
2.9.2 cl_closed . . . 140
2.9.3 cl_compare . . . 141
2.9.4 cl_multiclass . . . 142
2.9.5 cl_scheduling . . . 143
2.9.6 cl_stations . . . 144
2.9.7 cl_sweep . . . 145
2.9.8 dispatch_closed . . . 146
2.9.9 dispatch_mixed . . . 147
2.9.10 dispatch_open . . . 148
2.10 Stochastic Petri nets . . . 149
2.10.1 spn_basic_closed . . . 149
2.10.2 spn_basic_open . . . 150
2.10.3 spn_closed_fourplaces . . . 151
2.10.4 spn_closed_twoplaces . . . 152
2.10.5 spn_fluid_dae . . . 154
2.10.6 spn_fourmodes . . . 155
2.10.7 spn_inhibiting . . . 156
2.10.8 spn_lpbounds . . . 158
2.10.9 spn_open_sevenplaces . . . 159
2.10.10 spn_pareto_service . . . 160
2.10.11 spn_productform_nc . . . 161
2.10.12 spn_queueing_place . . . 163
2.10.13 spn_twomodes . . . 164
2.10.14 test_spn_nrm_open . . . 165
2.11 Workflow models . . . 166
2.11.1 wf_aph . . . 166
2.11.2 wf_branch . . . 167
2.11.3 wf_complex . . . 167
2.11.4 wf_erlang . . . 168
2.11.5 wf_loop . . . 169
2.11.6 wf_parallel . . . 169
2.11.7 wf_serial . . . 170

3 Advanced models **172**
3.1 Load-dependent stations . . . 172
3.1.1 fes_aggregation . . . 172
3.1.2 fes_single_class . . . 174
3.1.3 ld_class_dependence . . . 175
3.1.4 ld_fes_multiclass . . . 176
3.1.5 ld_fes_singleclass . . . 178
3.1.6 ld_global_dependence . . . 179
3.1.7 ld_joint_dependence . . . 180
3.1.8 ld_multiserver_fcfs . . . 182
3.1.9 ld_multiserver_ps . . . 183
3.1.10 ld_multiserver_ps_twoclasses . . . 184
3.1.11 ld_whittle_bandwidth . . . 186
3.2 Finite capacity regions . . . 187
3.2.1 fcr_constraints . . . 187
3.2.2 fcr_lincon . . . 188
3.2.3 fcr_lossn . . . 189
3.2.4 fcr_mm1kdrop . . . 191
3.2.5 fcr_mm1waitq . . . 192
3.2.6 fcr_oqndrop . . . 193
3.2.7 fcr_oqnwaitq . . . 194
3.3 State probabilities . . . 196
3.3.1 statepr_aggr . . . 196
3.3.2 statepr_aggr_large . . . 197
3.3.3 statepr_allprobs_fcfs . . . 198
3.3.4 statepr_allprobs_ps . . . 200
3.3.5 statepr_sys_aggr . . . 201
3.3.6 statepr_sys_aggr_large . . . 203
3.3.7 statepr_sys_marg . . . 204
3.4 Initial state and warm start . . . 206
3.4.1 init_state_fcfs_exp . . . 206
3.4.2 init_state_fcfs_nonexp . . . 207
3.4.3 init_state_ps . . . 209
3.4.4 ldes_warmstart . . . 210
3.5 State-dependent routing . . . 212
3.5.1 sdroute_closed . . . 212
3.5.2 sdroute_jsq . . . 213
3.5.3 sdroute_krzesinski . . . 215
3.5.4 sdroute_multibranch . . . 216
3.5.5 sdroute_open . . . 217
3.5.6 sdroute_twoclasses_closed . . . 219
3.6 Cyclic polling . . . 220
3.6.1 polling_exhaustive_det . . . 220
3.6.2 polling_exhaustive_exp . . . 222
3.6.3 polling_gated . . . 223
3.6.4 polling_klimited . . . 224
3.7 Switchover times . . . 226
3.7.1 switchover_basic . . . 226
3.8 Response time distribution . . . 227
3.8.1 cdf_respt_closed . . . 227

3.8.2 cdf_respt_closed_threeclasses . . . 229
3.8.3 cdf_respt_distrib . . . 230
3.8.4 cdf_respt_open_twoclasses . . . 231
3.8.5 cdf_respt_populations . . . 233
3.9 Busy periods . . . 234
3.9.1 busyp_subnetwork . . . 234
3.10 Stochastic network calculus . . . 236
3.10.1 snc_delay_quantile . . . 236
3.10.2 snc_tandem_multiclass . . . 237
3.11 Random environments . . . 238
3.11.1 example_mapqn2renv . . . 239
3.11.2 renv_basic . . . 240
3.11.3 renv_container_terminal . . . 241
3.11.4 renv_fourstages_repairmen . . . 243
3.11.5 renv_genqn . . . 244
3.11.6 renv_lqn_twostages . . . 245
3.11.7 renv_map_fallback . . . 246
3.11.8 renv_node_breakdown . . . 247
3.11.9 renv_rotterdam_blending . . . 248
3.11.10 renv_threestages_repairmen . . . 249
3.11.11 renv_twostages_repairmen . . . 250
3.12 Layered cache-queueing models . . . 252
3.12.1 lcq_singlehost . . . 252
3.12.2 lcq_threehosts . . . 253
3.13 Pass and swap . . . 255
3.13.1 pas_closed_tandem_fig6 . . . 255
3.13.2 pas_compatibility_5class . . . 256
3.13.3 pas_cyclic_ctmc_vs_bruteforce . . . 258
3.13.4 pas_cyclic_normconst_bruteforce . . . 259
3.13.5 pas_mmk . . . 260
3.13.6 pas_nc_sampling . . . 261
3.13.7 pas_saturation . . . 261
3.13.8 pas_selfloop . . . 263
3.14 Reward models . . . 264
3.14.1 rewardModel_aggregation . . . 264
3.14.2 rewardModel_mm1k . . . 265
3.14.3 rewardModel_multiclass . . . 267
3.14.4 rewardModel_templates . . . 268
3.15 Agent models . . . 269
3.15.1 ag_closed_network . . . 269
3.15.2 ag_gnetwork . . . 271
3.15.3 ag_jackson_network . . . 272
3.15.4 ag_multiclass_closed . . . 273
3.15.5 ag_tandem_open . . . 274
3.15.6 ag_tandem_phasetype . . . 276
3.16 Large-scale models . . . 277
3.16.1 largescale_tbi . . . 277
3.17 Custom solvers . . . 278
3.17.1 solver_custom . . . 278
3.17.2 solver_custom_analyzer . . . 278

4 Model gallery **280**
4.1 Reading the names . . . 280
4.2 The catalogue . . . 281
4.2.1 gallery_aphm1 . . . 281
4.2.2 gallery_cache_lru . . . 282
4.2.3 gallery_cache_routing . . . 283
4.2.4 gallery_coxm1 . . . 284
4.2.5 gallery_cqn . . . 285
4.2.6 gallery_cqn_multiclass . . . 285
4.2.7 gallery_detm1 . . . 286
4.2.8 gallery_dm1 . . . 287
4.2.9 gallery_erldk . . . 288
4.2.10 gallery_erlerl1 . . . 288
4.2.11 gallery_erlerl1_reentrant . . . 289
4.2.12 gallery_erlm1 . . . 290
4.2.13 gallery_erlm1_ps . . . 291
4.2.14 gallery_erlm1_reentrant . . . 291
4.2.15 gallery_erlm1ps . . . 292
4.2.16 gallery_fcr . . . 293
4.2.17 gallery_fj_closed . . . 293
4.2.18 gallery_fj_open . . . 294
4.2.19 gallery_fj_quorum . . . 295
4.2.20 gallery_gamm1 . . . 297
4.2.21 gallery_hyperl1_feedback . . . 297
4.2.22 gallery_hyperl1_reentrant . . . 298
4.2.23 gallery_hyperlk . . . 299
4.2.24 gallery_hyphyp1_linear . . . 300
4.2.25 gallery_hyphyp1_reentrant . . . 300
4.2.26 gallery_hyphyp1_tandem . . . 301
4.2.27 gallery_hypm1 . . . 302
4.2.28 gallery_hypm1_reentrant . . . 303
4.2.29 gallery_lqn_basic . . . 303
4.2.30 gallery_lqn_random . . . 304
4.2.31 gallery_lqn_workflows . . . 305
4.2.32 gallery_lukumar_reentrant . . . 306
4.2.33 gallery_mapm1 . . . 308
4.2.34 gallery_mapmk . . . 308
4.2.35 gallery_mdk . . . 309
4.2.36 gallery_merl1 . . . 310
4.2.37 gallery_merl1_linear . . . 310
4.2.38 gallery_merl1_reentrant . . . 311
4.2.39 gallery_merl1_tandem . . . 312
4.2.40 gallery_merlk . . . 313
4.2.41 gallery_mhyp1 . . . 313
4.2.42 gallery_mhyp1_linear . . . 314
4.2.43 gallery_mhyp1_reentrant . . . 315
4.2.44 gallery_mhyp1_tandem . . . 316
4.2.45 gallery_mhypk . . . 316
4.2.46 gallery_mm1 . . . 317
4.2.47 gallery_mm1_feedback . . . 317

4.2.48 gallery_mm1_linear . . . 318
4.2.49 gallery_mm1_multiclass . . . 319
4.2.50 gallery_mm1_prio . . . 320
4.2.51 gallery_mm1_ps . . . 321
4.2.52 gallery_mm1_ps_feedback . . . 321
4.2.53 gallery_mm1_ps_multiclass . . . 322
4.2.54 gallery_mm1_ps_reentrant . . . 323
4.2.55 gallery_mm1_reentrant . . . 324
4.2.56 gallery_mm1_tandem . . . 324
4.2.57 gallery_mm1_tandem_multiclass . . . 325
4.2.58 gallery_mm1k . . . 326
4.2.59 gallery_mmap1 . . . 327
4.2.60 gallery_mmap1_multiclass . . . 327
4.2.61 gallery_mmapk . . . 328
4.2.62 gallery_mmk . . . 329
4.2.63 gallery_mpar1 . . . 330
4.2.64 gallery_multitier . . . 330
4.2.65 gallery_multitier_storage . . . 331
4.2.66 gallery_parm1 . . . 333
4.2.67 gallery_qn_random . . . 334
4.2.68 gallery_renv_breakdown . . . 335
4.2.69 gallery_repairmen . . . 335
4.2.70 gallery_replayerm1 . . . 336
4.2.71 gallery_um1 . . . 337

5 Discrete-time models **339**
5.1 The slotted models . . . 339
5.1.1 dt_bernoulli_loaddep . . . 339
5.1.2 dt_cycle . . . 340
5.1.3 dt_cycle_loaddep . . . 341
5.1.4 dt_cycle_multiclass . . . 342
5.1.5 dt_geogeo1 . . . 343
5.1.6 dt_geogeo1_loss . . . 344

6 Solver-specific examples **346**
6.0.1 ctmc_spn_mm1 . . . 346
6.0.2 ctmc_spn_vs_nc_closed_qn . . . 347
6.0.3 ctmc_tandem_mm1 . . . 349

7 Model interchange **351**
7.1 The fixtures . . . 351
7.1.1 json_balking_retrial . . . 351
7.1.2 json_classcap_droprule . . . 352
7.1.3 json_fcr_details . . . 353
7.1.4 json_hetero_servers . . . 354
7.1.5 json_join_deadline . . . 354
7.1.6 json_signal_classes . . . 356

8 Inference and estimation **358**
8.1 The estimators . . . 358
8.1.1 est_ekf_changepoint . . . 358
8.1.2 est_ekf_closed . . . 359
8.1.3 est_ekf_open . . . 361
8.1.4 est_erps_closed . . . 362
8.1.5 est_erps_open . . . 363
8.1.6 est_mcmc_closed . . . 364
8.1.7 est_mle_open . . . 365
8.1.8 est_qmle_closed . . . 367
8.1.9 est_trace_closed . . . 368
8.1.10 est_ubo_changepoint . . . 369
8.1.11 est_ubo_closed . . . 370
8.1.12 est_ubo_open . . . 371
8.1.13 est_ubo_variants_closed . . . 372
8.1.14 est_ubr_changepoint . . . 374
8.1.15 est_ubr_closed . . . 375
8.1.16 est_ubr_open . . . 376
8.1.17 est_vi_closed . . . 377

9 Optimization **379**
9.1 Sizing an M/M/c queue . . . 379
9.2 Exact sizing by bisection . . . 380
9.3 Continuous service rate tuning . . . 380
9.4 Load balancing across heterogeneous servers . . . 381
9.5 Population sizing in a closed network . . . 381
9.6 Robust sizing under workload scenarios . . . 382
9.7 Cost-performance tradeoff frontier . . . 382
9.8 Decomposing a multi-variable problem . . . 382
9.9 The models behind the tutorials . . . 383
9.9.1 opt_bisection_sizing . . . 383
9.9.2 opt_decomposition . . . 384
9.9.3 opt_load_balancing . . . 386
9.9.4 opt_lqn_host_demand . . . 387
9.9.5 opt_pareto_frontier . . . 388
9.9.6 opt_population_sizing . . . 390
9.9.7 opt_robust_sizing . . . 391
9.9.8 opt_sensitivity_report . . . 393
9.9.9 opt_server_sizing . . . 394
9.9.10 opt_service_rate . . . 395

Chapter 1

About this manual

This manual documents the worked-example inventory of LINE, one edition per codebase. An entry is runnable only when this edition identifies a source implementation and does not mark the required operation as unsupported: `python/examples/`, and each chapter follows the organization of that tree. Script names agree across the four codebases, so an example can be read side by side between editions; where a codebase does not carry a family at all, the chapter says so rather than leaving the reader to infer it.

The reference material on model classes, solvers and options is in the LINE user manual, and the guided tour and the introductory tutorials are in the LINE primer. Neither is repeated here: this manual starts where the primer stops, at the point where one already knows how to build a model and wants to see the feature one needs.

Every model in the suite is written in the same four blocks, and it is worth learning to read them:

1. the nodes: stations, sources, sinks and the special nodes;
2. the classes and their arrival and service distributions;
3. the topology, as a routing matrix or a serial routing shorthand;
4. the solution: one or more solvers, and the metrics read from them.

The chapters are:

- *Basic models* (Chapter 2): the elementary model families, one per node or class feature.
- *Advanced models* (Chapter 3): features that leave the product-form setting.
- *Model gallery* (Chapter 4): a catalogue of small named models, in Kendall notation where one applies.
- *Discrete-time models* (Chapter 5): models on a slotted time scale.
- *Solver-specific examples* (Chapter 6): scripts written around one solver rather than one model.
- *Model interchange* (Chapter 7): the JSON model format and its round-trip fixtures.
- *Inference and estimation* (Chapter 8): fitting model parameters to measurements.
- *Optimization* (Chapter 9): design problems on queueing networks.

Each chapter opens with a worked example and then gives one section per script in the reference inventory. A complete entry contains a topology figure, a feature table, source code, and captured output. These components are generated from the harvested model, the implementation in the named language tree, and an actual run.

Chapter 2

Basic models

The basic examples cover one model family each: a node type, a class type or a scheduling feature, in the smallest model that shows it. They live under `python/examples/basic/`, one subdirectory per family, and the script names agree across the four codebases, so a model can be compared line by line between editions.

2.1 Open queueing networks

An open network has an unbounded population: jobs enter at a `Source`, visit stations according to the routing matrix and leave at a `Sink`. Stability requires the utilization of every station to stay below one, which is a property of the arrival rates and the demands, not something the model declares.

The archetype is `oqn_basic`: a Poisson stream feeds a Delay station with hyperexponential think time and an FCFS queue with exponential service. Four blocks build it – nodes, classes and their distributions, the routing matrix, and the solve – and every model in this manual follows the same four blocks.

```
model = Network('model')
# Block 1: nodes
node = np.empty(4, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
# Block 2: classes
jobclass = OpenClass(model, 'Class1', 0)
node[0].set_service(jobclass, HyperExp(0.5, 3.0, 10.0))
node[1].set_service(jobclass, Exp(1))
node[2].set_arrival(jobclass, Exp(0.1))
# Block 3: topology
P = model.init_routing_matrix()
pmatrix = [[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 0, 0]]
for i in range(len(node)):
    for j in range(len(node)):
        P.set(jobclass, jobclass, node[i], node[j], pmatrix[i][j])
model.link(P)
# Block 4: solution
print(MVA(model).avg_table())
```

The scripts of this family are:

2.1.1 `oqn_basic`

Source, Delay and an FCFS queue with one open class and a hyperexponential think time, solved with every solver that admits the model.

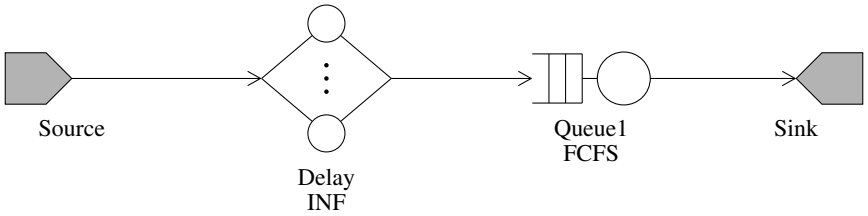

Figure 2.1: Topology of `oqn_basic`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	1: Class1 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(0.1)
Service	Delay – Class1: HyperExp(0.5,0.5;3,10) Queue1 – Class1: Exp(1)
Solvers	CTMC, FLD, MVA, MAM, NC, JMT, SSA, LDES

Table 2.1: Features of `oqn_basic`.

The example is built and solved as follows.

```
model = Network('model')
node = np.empty(4, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = OpenClass(model, 'Class1', 0)
node[0].set_service(jobclass, HyperExp(0.5, 3.0, 10.0))
node[1].set_service(jobclass, Exp(1))
node[2].set_arrival(jobclass, Exp(0.1))
P = model.init_routing_matrix()
pmatrix = [[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 0, 0]]
for i in range(len(node)):
    for j in range(len(node)):
        P.set(jobclass, jobclass, node[i], node[j], pmatrix[i][j])
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)

# The model is built by the function above, not inline: an example that
# builds it only under __main__ exposes nothing on import, so the JAVA and
# C++ rows of parity-static cannot export it and SKIP every solver. That
# reads as coverage while asserting nothing (see _example_model_vendor.py).
model = oqn_basic()

# Run multiple solvers

    ... (15 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
```

```
    t>s.
Station  JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.021667  0.021667  0.21667  0.21667   0.1   0.1
Queue1   Class1     0.11111       0.1   1.1111   1.1111   0.1   0.1
Source   Class1           0         0        0        0     0   0.1

SOLVER: Fluid
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
    completed in <t>s.
Station  JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.021667  0.021667  0.21667  0.21667   0.1   0.1
Queue1   Class1     0.10021       0.1   1.0021   1.0021   0.1   0.1

... (41 captured-output lines omitted; rerun the source example for the full output)

Source   Class1           0         0        0        0         0       0.1

SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station  JobClass      QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.021461  0.021697  0.21425  0.21425  0.10335  0.10014
Queue1   Class1     0.11138   0.10026   1.1107   1.1107  0.10335  0.10014
Source   Class1           0         0        0        0        0      0.1
```

The rows repeat for each of the 8 solvers the script runs (CTMC, FLD, MVA, MAM, NC, JMT, SSA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.1.2 `oqn_cs_routing`

Three open classes over two PS queues, two of which switch class at a `ClassSwitch` node; the topology is imported from a JSIMgraph file.

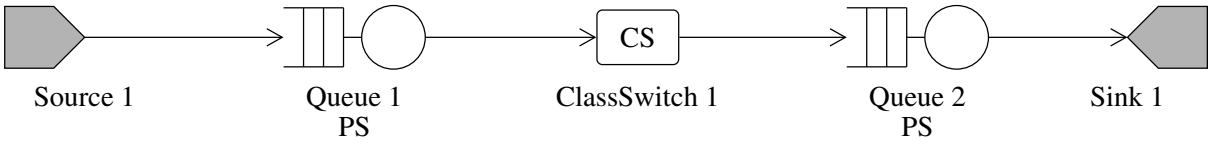

Figure 2.2: Topology of `oqn_cs_routing`: an open network over 2 queueing stations with a class-switch node.

Nodes	5: Source, Queue × 2, ClassSwitch, Sink	
Classes	3: Class A (open), Class B (open), Class C (open)	
Scheduling	Queue 1: PS; Queue 2: PS	
Arrivals	Source 1 – Class A: Exp(2); Class B: Exp(1); Class C: Disabled	
Service	Queue 1 – Class A: Exp(5); Class B: Exp(3.333); Class C: Exp(3)	Queue 2 – Class A: Exp(1); Class B: Exp(1); Class C: Exp(6.667)
Solvers	CTMC, FLD, MVA, MAM, NC, JMT, SSA, LDES	

Table 2.2: Features of `oqn_cs_routing`.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('myModel')

# Nodes
node = np.empty(5, dtype=object)
node[0] = Source(model, 'Source 1')
node[1] = Queue(model, 'Queue 1', SchedStrategy.PS)
node[2] = ClassSwitch(model, 'ClassSwitch 1')
node[3] = Sink(model, 'Sink 1')
node[4] = Queue(model, 'Queue 2', SchedStrategy.PS)
```

```
# Classes
jobclass = np.empty(3, dtype=object)
jobclass[0] = OpenClass(model, 'Class A', 0)
jobclass[1] = OpenClass(model, 'Class B', 0)
jobclass[2] = OpenClass(model, 'Class C', 0)

# Arrivals
node[0].set_arrival(jobclass[0], Exp.fit_mean(0.5))
node[0].set_arrival(jobclass[1], Exp.fit_mean(1.0))
node[0].set_arrival(jobclass[2], Disabled.get_instance())

# Service times
node[1].set_service(jobclass[0], Exp.fit_mean(0.2))
node[1].set_service(jobclass[1], Exp.fit_mean(0.3))

    ... (46 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = [[1, 1, 0], [3, 3, 0], [0, 0, 3]] for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source 1  Class A         0        0        0        0        0   1.8159
Source 1  Class B         0        0        0        0        0  0.94121
Queue 1   Class A    0.8844  0.36317  0.48704  0.32469   1.8159   1.8159
Queue 1   Class B   0.71089  0.28236  0.75529  0.25176  0.94121  0.94121
Queue 2   Class C   0.61834  0.41356  0.22427  0.22427   2.7571   2.7571

SOLVER: Fluid
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
    completed in <t>s.
Station   JobClass     QLen  Util    RespT   ResidT  ArvR  Tput

... (59 captured-output lines omitted; rerun the source example for the full output)

SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 100000 --seed 23000
Station   JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source 1  Class A         0        0        0        0        0        2
Source 1  Class B         0        0        0        0        0        1
Source 1  Class C         0        0        0        0        0      NaN
Queue 1   Class A    1.3086  0.39838  0.65708  0.43805   1.9923   1.9901
Queue 1   Class B   0.96972  0.29936   0.9717   0.3239  0.99805  0.99703
Queue 2   Class C   0.81451  0.44816   0.2726   0.2726   2.9901   2.9868
```

The rows repeat for each of the 8 solvers the script runs (CTMC, FLD, MVA, MAM, NC, JMT, SSA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.1.3 `oqn_fourqueues`

A larger multiclass network over four stations mixing PS and FCFS scheduling.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('MyNetwork')

# Nodes
node = np.empty(6, dtype=object)
node[0] = Source(model, 'Source')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Queue(model, 'Queue2', SchedStrategy.FCFS)
node[3] = Queue(model, 'Queue3', SchedStrategy.PS)
node[4] = Queue(model, 'Queue4', SchedStrategy.FCFS)
node[5] = Sink(model, 'Sink')

# Classes
jobclass = np.empty(3, dtype=object)
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
jobclass[2] = OpenClass(model, 'Class3', 0)

# Arrivals
node[0].set_arrival(jobclass[0], Exp.fit_mean(5.0))
node[0].set_arrival(jobclass[1], Exp.fit_mean(8.0))
node[0].set_arrival(jobclass[2], Exp.fit_mean(7.0))

# Service times for Class1
node[1].set_service(jobclass[0], Exp.fit_mean(0.3))

    ... (58 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.1.4 `oqn_mapt`

A time-inhomogeneous MAP arrival stream and the Ko-Pender fluid and diffusion limits returned by the `kp` fluid method.

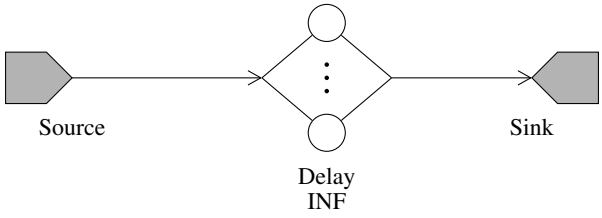

Figure 2.3: Topology of `oqn_mapt`: an open network over 1 queueing station.

The example is built and solved as follows.

```
                    Source)
model = Network('model')
source = Source(model, 'Source')
delay = Delay(model, 'Delay')
sink = Sink(model, 'Sink')
jobclass = OpenClass(model, 'OpenClass', 0)
# Two segments of a 2-phase MAP, held for 1 and 1.5 time units and repeating.
```

Nodes	3: Source, Delay, Sink
Classes	1: OpenClass (open)
Scheduling	Delay: INF
Arrivals	Source – OpenClass: MAPt
Service	Delay – OpenClass: Exp(2)
Solvers	FLD

Table 2.3: Features of `oqn_mapt`.

```
# The second segment runs the same phase graph at roughly twice the rate.
breakpoints = [0.0, 1.0, 2.5]
D0 = [np.array([[-5.0, 1.0], [2.0, -4.0]]),
      np.array([[-12.0, 3.0], [5.0, -9.0]])]
D1 = [np.array([[3.0, 1.0], [1.0, 1.0]]),
      np.array([[7.0, 2.0], [2.0, 2.0]])]
arrival = MAPt(breakpoints, D0, D1, True)
source.setArrival(jobclass, arrival)
delay.setService(jobclass, Exp(2.0))
model.link(Network.serialRouting(source, delay, sink))
# The arrival process rides along: the schedule is what the transient block
# probes, and find_model searches a returned tuple element-wise, so handing
# both back keeps the model exportable.
return model, arrival
# Built by the function above rather than inline: a model that exists only
# under __main__ exposes nothing on import, so the JAVA and C++ parity rows
# cannot export it and SKIP every solver -- a row that reads as coverage
# while asserting nothing (see parity-static/_example_model_vendor.py).
model, arrival = oqn_mapt()

    ... (14 source lines omitted; full implementation: python/examples/basic/)

print('\n   t     lambda(t)    QLen mean     QLen var')
for probe in (0.5, 1.0, 1.5, 2.5, 3.0, 4.0, 5.0):
    print('%6.2f  %10.4f  %11.6f  %11.6f'
          % (probe, arrival.getRateAt(probe),
             float(np.interp(probe, t, mean)),
             float(np.interp(probe, t, var))))
```

Running this edition prints

```
Fluid analysis [method: kp; type: approximate, deterministic; lang: python; env: python] completed in <t>s
     .
Station  JobClass     QLen    Util  RespT  ResidT    ArvR    Tput
Source   OpenClass       0       0      0       0       0  5.4301
Delay    OpenClass  2.7151  2.7151    0.5     0.5  5.4301  5.4301
Fluid analysis [method: kp; type: approximate, deterministic; lang: python; env: python] completed in <t>s
     .

   t     lambda(t)    QLen mean     QLen var
  0.50      3.2000     1.011269     1.040242
  1.00      6.9167     1.383464     1.421803
  1.50      6.9167     2.698003     2.834981
  2.50      3.2000     3.355447     3.512559
  3.00      3.2000     2.242813     2.303353
  4.00      6.9167     2.864517     3.002174
  5.00      3.2000     3.377981     3.535106
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `FLD`.

2.1.5 `oqn_multichain_cs`

Five classes grouped into three chains by class switching, the smallest model in which chains and classes differ.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow)

or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
model = Network('oqn_multichain')

# Block 1: nodes
node = np.empty(6, dtype=object)
node[0] = Source(model, 'RequestSource')
node[1] = Queue(model, 'Router', SchedStrategy.FCFS)
node[2] = Queue(model, 'WebCache', SchedStrategy.PS)
node[3] = Queue(model, 'AppServer', SchedStrategy.FCFS)
node[4] = Queue(model, 'DataServer', SchedStrategy.PS)
node[5] = Sink(model, 'ResponseSink')

# Block 2: job classes
# Chain 1: Interactive requests (can switch between InteractiveA and InteractiveB)
jobclass = np.empty(5, dtype=object)
jobclass[0] = OpenClass(model, 'InteractiveA', 0)
jobclass[1] = OpenClass(model, 'InteractiveB', 0)
# Chain 2: Batch jobs (can switch between BatchA and BatchB)
jobclass[2] = OpenClass(model, 'BatchA', 0)
jobclass[3] = OpenClass(model, 'BatchB', 0)
# Chain 3: Real-time tasks
jobclass[4] = OpenClass(model, 'RealTime', 0)

# Block 3: service times
# Router
node[1].set_service(jobclass[0], Exp.fit_mean(0.15))
node[1].set_service(jobclass[1], Exp.fit_mean(0.18))

    ... (89 source lines omitted; full implementation: python/examples/basic/)


for s in range(len(solver)):
    print(f'SOLVER: {solver[s].get_name()}')
    avg_table = solver[s].avg_table()
    print(avg_table)
    print()
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.1.6 oqn_nhpp

A cyclic non-homogeneous Poisson arrival stream, whose rate is a periodic function of time.

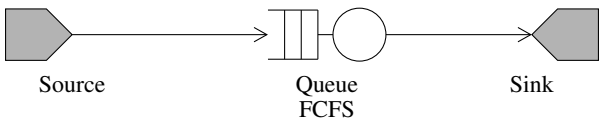

Figure 2.4: Topology of oqn_nhpp: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: OpenClass (open)
Scheduling	Queue: FCFS
Arrivals	Source – OpenClass: NHPP
Service	Queue – OpenClass: Exp(10)
Solvers	LDES

Table 2.4: Features of oqn_nhpp.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')

jobclass = OpenClass(model, 'OpenClass', 0)

# Rates 2,8,4 held for 3,1,2 time units, repeating cyclically.
source.set_arrival(jobclass, NHPP([0, 3, 4, 6], [2, 8, 4], True))
queue.set_service(jobclass, Exp(10))

model.link(Network.serial_routing([source, queue, sink]))

avg_table_1 = LDES(model, seed=1234, samples=100000).avg_table()
print('DES Result:')
print(avg_table_1)

# Fluid transient over two periods: the queue throughput tracks lambda(t).
nhpp = source.getArrivalProcess(jobclass)
_, _, TNt = SolverFLD(model, timespan=[0, 12], verbose=0).getTranAvg()
t = np.asarray(TNt[1][0].t).ravel()
y = np.asarray(TNt[1][0].metric).ravel()
print('FLD transient:')
for tt in [1.5, 3.5, 5.0, 7.5, 9.5, 11.0]:
    print('  t=%5.2f lambda(t)=%6.3f queue Tput=%6.3f'
          % (tt, nhpp.getRateAt(tt), float(np.interp(tt, t, y))))
```

Running this edition prints

```
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 100000 --seed 1234
DES Result:
Station  JobClass      QLen     Util     RespT    ResidT    ArvR     Tput
Source   OpenClass        0        0         0         0       0   3.6667
Queue    OpenClass   0.7244  0.36847   0.19681   0.19681  3.6846   3.6809
FLD transient:
  t= 1.50 lambda(t)= 2.000 queue Tput= 2.000
  t= 3.50 lambda(t)= 8.000 queue Tput= 7.959
  t= 5.00 lambda(t)= 4.000 queue Tput= 4.000
  t= 7.50 lambda(t)= 2.000 queue Tput= 2.000
  t= 9.50 lambda(t)= 8.000 queue Tput= 7.959
  t=11.00 lambda(t)= 4.000 queue Tput= 4.000
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `LDES`.

2.1.7 `oqn_oneline`

The same tandem written in one line from the arrival, demand and think time matrices, through `Network.tandemPsInf`.

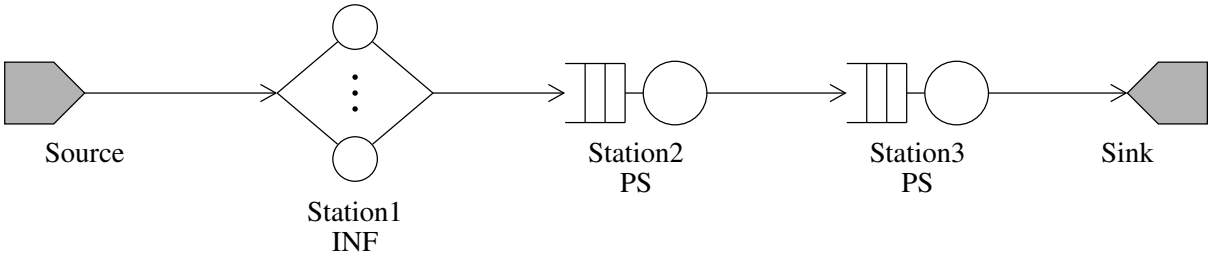

Figure 2.5: Topology of `oqn_oneline`: an open network over 3 queueing stations.

Nodes	5: Source, Queue × 2, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Station1: INF; Station2: PS; Station3: PS
Arrivals	Source – Class1: Exp(0.02); Class2: Exp(0.04)
Service	Station1 – Class1: Exp(0.01099); Class2: Exp(0.01087) Station2 – Class1: Exp(0.1); Class2: Exp(0.2) Station3 – Class1: Exp(0.2); Class2: Exp(0.1111)
Solvers	MVA

Table 2.5: Features of `oqn_oneline`.

The example is built and solved as follows.

```
print('This example shows a compact solution of a tandem open queueing network.')

# Model parameters
lambda_rates = np.array([1, 2]) / 50  # lambda(r) - arrival rate of class r
D = np.array([[10, 5], [5, 9]])  # D(i,r) - mean demand of class r at station i
Z = np.array([91, 92])  # Z(r) - mean service time of class r at delay station

# Create tandem PS network with infinite servers
model = Network.tandem_ps_inf(lambda_rates, D, Z)

# Solve with MVA
solver = MVA(model)
avg_table = solver.avg_table()
print(avg_table)

# 1-line version: avg_table = MVA(Network.tandem_ps_inf(lambda_rates, D, Z)).avg_table()
```

Running this edition prints

```
This example shows a compact solution of a tandem open queueing network.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   JobClass     QLen  Util   RespT  ResidT  ArvR  Tput
Source    Class1          0     0       0       0     0  0.02
Source    Class2          0     0       0       0     0  0.04
Station1  Class1       1.82  1.82      91      91  0.02  0.02
Station1  Class2       3.68  3.68      92      92  0.04  0.04
Station2  Class1    0.33333   0.2  16.667  16.667  0.02  0.02
Station2  Class2    0.33333   0.2  8.3333  8.3333  0.04  0.04
Station3  Class1    0.18518   0.1  9.2592  9.2592  0.02  0.02
Station3  Class2    0.66667  0.36  16.667  16.667  0.04  0.04
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.1.8 `oqn_trace_driven`

Arrivals and service times replayed from a measurement trace with `Replayer`.

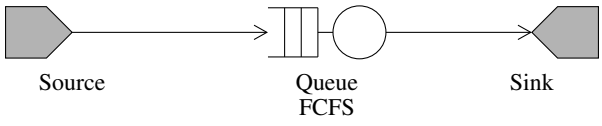

Figure 2.6: Topology of `oqn_trace_driven`: an open network over 1 queueing station.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

source = Source(model, 'Source')
```

Nodes	3: Source, Queue, Sink
Classes	1: OpenClass (open)
Scheduling	Queue: FCFS
Arrivals	Source – OpenClass: Exp(1)
Service	Queue – OpenClass: Replayer(trace)
Solvers	JMT, LDES

Table 2.6: Features of `oqn_trace_driven`.

```
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')

jobclass = OpenClass(model, 'OpenClass', 0)

source.set_arrival(jobclass, Exp(1))

# The trace ships with the MATLAB twin of this example
here = os.path.dirname(os.path.abspath(__file__))
trace_file = os.path.join(here, '..', '..', '..', '..', 'matlab', 'examples',
                          'basic', 'openQN', 'example_trace.txt')
queue.set_service(jobclass, Replayer(os.path.normpath(trace_file)))

model.link(Network.serial_routing([source, queue, sink]))

# Run solvers
avg_table_1 = JMT(model, seed=23000).avg_table()
print('JMT Solver:')
print(avg_table_1)

avg_table_2 = LDES(model, seed=23000).avg_table()
print('\nLDES Solver:')
print(avg_table_2)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
JMT Solver:
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
Source   OpenClass       0        0        0        0       0  1.0205
Queue    OpenClass  0.1128  0.10284  0.11275  0.11275  1.0205  1.0211
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000

LDES Solver:
Station  JobClass      QLen     Util    RespT   ResidT    ArvR    Tput
Source   OpenClass        0        0        0        0       0       1
Queue    OpenClass  0.11241  0.10112  0.11236  0.11236  1.0014  1.0004
```

The rows repeat for each of the 2 solvers the script runs (JMT, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.1.9 `oqn_vsinks`

Several sinks in one model, the virtual-sink idiom for measuring departure streams separately.

Nodes	5: Source, Queue, Router × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Exp(1)
Service	Queue1 – Class1: Exp(100); Class2: Exp(100)
Solvers	MVA, MAM, NC

Table 2.7: Features of `oqn_vsinks`.

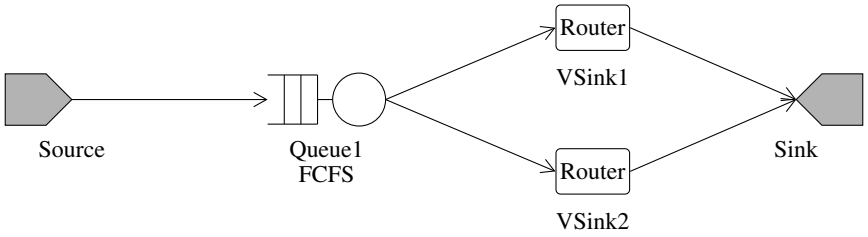

Figure 2.7: Topology of `oqn_vsinks`: an open network over 1 queueing station with a router.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

source = Source(model, 'Source')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
vsink1 = Router(model, 'VSink1')
vsink2 = Router(model, 'VSink2')

ocl1 = OpenClass(model, 'Class1')
ocl2 = OpenClass(model, 'Class2')

source.set_arrival(ocl1, Exp(1.0))
queue.set_service(ocl1, Exp(100.0))

source.set_arrival(ocl2, Exp(1.0))
queue.set_service(ocl2, Exp(100.0))

P = model.init_routing_matrix()

# Class 1 routing (60% to vsink1, 40% to vsink2)
P.set(ocl1, ocl1, source, queue, 1.0)
P.set(ocl1, ocl1, queue, vsink1, 0.6)
P.set(ocl1, ocl1, queue, vsink2, 0.4)
P.set(ocl1, ocl1, vsink1, sink, 1.0)

    ... (29 source lines omitted; full implementation: python/examples/basic/)

avg_table_nc = NC(model).avg_table()
print(avg_table_nc)

avg_node_table_nc = NC(model).avg_node_table()
print('\nNC Node-level metrics:')
print(avg_node_table_nc)
```

Running this edition prints

```
MVA Results:
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station  JobClass      QLen  Util     RespT    ResidT  ArvR  Tput
Source   Class1          0     0         0         0     0     1
Source   Class2          0     0         0         0     0     1
Queue1   Class1   0.010204  0.01  0.010204  0.010204     1     1
Queue1   Class2   0.010204  0.01  0.010204  0.010204     1     1
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

Node-level metrics (including virtual sinks):
     Node JobClass    QLen  Util   RespT  ResidT  ArvR  Tput
0  Source   Class1  0.0000  0.00  0.0000  0.0000   0.0   1.0
1  Source   Class2  0.0000  0.00  0.0000  0.0000   0.0   1.0
2  Queue1   Class1  0.0102  0.01  0.0102  0.0102   1.0   1.0
3  Queue1   Class2  0.0102  0.01  0.0102  0.0102   1.0   1.0
4    Sink   Class1  0.0000  0.00  0.0000  0.0000   1.0   0.0
```

```
... (41 captured-output lines omitted; rerun the source example for the full output)

2  Queue1   Class1  0.0102  0.01  0.0102  0.0102   1.0   1.0
3  Queue1   Class2  0.0102  0.01  0.0102  0.0102   1.0   1.0
4    Sink   Class1  0.0000  0.00  0.0000  0.0000   1.0   0.0
5    Sink   Class2  0.0000  0.00  0.0000  0.0000   1.0   0.0
6  VSink1   Class1  0.0000  0.00  0.0000  0.0000   0.6   0.6
7  VSink1   Class2  0.0000  0.00  0.0000  0.0000   0.1   0.1
8  VSink2   Class1  0.0000  0.00  0.0000  0.0000   0.4   0.4
9  VSink2   Class2  0.0000  0.00  0.0000  0.0000   0.9   0.9
```

The rows repeat for each of the 3 solvers the script runs (MVA, MAM, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2 Closed queueing networks

A closed network holds a fixed population per class. Each closed class names a reference station, at which its visit ratio is one and through which its throughput is measured; the population is the third argument of the class constructor. A Delay station with think time Z and a queue with demand D is the interactive-system archetype, and the repairman model is its oldest form.

Because the population is finite, every metric is bounded and the network is stable by construction. What varies across the scripts below is the scheduling discipline, the service distribution and the number of servers, that is, exactly the assumptions under which a product-form solution does or does not hold.

2.2.1 `cqn_bas_blocking`

Blocking-after-service with finite buffers, where a completing job holds its server until the destination frees a slot.

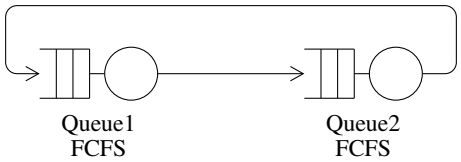

Figure 2.8: Topology of `cqn_bas_blocking`: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	1: Class1 (closed, $N = 2$)
Scheduling	Queue1: FCFS; Queue2: FCFS
Service	Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(0.8)
Features	drop rule {'Class1': 'blockingAfterService'} at Queue1; finite buffer 1 at Queue2
Solvers	MVA

Table 2.8: Features of `cqn_bas_blocking`.

The example is built and solved as follows.

```
model = Network('cqn_bas_blocking')
node = np.empty(2, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[1] = Queue(model, 'Queue2', SchedStrategy.FCFS)
jobclass = ClosedClass(model, 'Class1', 2, node[0], 0)
node[0].set_service(jobclass, Exp(1.0))
node[1].set_service(jobclass, Exp(0.8))
node[1].setCap(1)
node[0].setDropRule(jobclass, DropStrategy.BAS)
model.link(Network.serialRouting(node[0], node[1]))
return model
```

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = cqn_bas_blocking()

print('\nSOLVER: MVA (method=sqd)')
solver = MVA(model, 'sqd')
avg_table = solver.getAvgTable()
print(avg_table)
```

Running this edition prints

```
SOLVER: MVA (method=sqd)
MVA analysis [method: sqd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Class1    0.91777  0.54051   1.698   1.698  0.54051  0.54051
Queue2   Class1     1.0822  0.67563  2.0023  2.0023  0.54051  0.54051
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.2.2 `cqn_bcmp_theorem`

The product-form scheduling policies of the BCMP theorem (FCFS with class-independent exponential service, PS, LCFS-PR, INF) compared on one network.

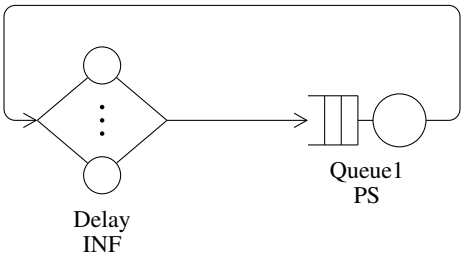

Figure 2.9: Topology of `cqn_bcmp_theorem`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Erlang(3,2); Class2: HyperExp(0.5,0.5;3,10) Queue1 – Class1: Exp(1); Class2: Exp(1)
Solvers	CTMC

Table 2.9: Features of `cqn_bcmp_theorem`.

The example is built and solved as follows.

```
c = 1
# PS scheduling model
ps_model = Network('PS scheduling model')
node = np.empty(2, dtype=object)
node[0] = Delay(ps_model, 'Delay')
node[1] = Queue(ps_model, 'Queue1', SchedStrategy.PS)
node[1].set_number_of_servers(c)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(ps_model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(ps_model, 'Class2', 2, node[0], 0)
node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))
node[1].set_service(jobclass[0], Exp(1))
node[1].set_service(jobclass[1], Exp(1))
P = ps_model.init_routing_matrix()
```

```
P[jobclass[0], jobclass[0]] = Network.serial_routing(node)
P[jobclass[1], jobclass[1]] = Network.serial_routing(node)
ps_model.link(P)
# FCFS scheduling model
fcfs_model = Network('FCFS scheduling model')
node = np.empty(2, dtype=object)
node[0] = Delay(fcfs_model, 'Delay')
node[1] = Queue(fcfs_model, 'Queue1', SchedStrategy.FCFS)
node[1].set_number_of_servers(c)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(fcfs_model, 'Class1', 2, node[0], 0)

    ... (34 source lines omitted; full implementation: python/examples/basic/)

solver = np.append(solver, CTMC(lcfspr_model))

for s in range(len(solver)):
    print(f'\nMODEL: {solver[s].model.get_name()}')
    avg_table = solver[s].avg_table()
    print(avg_table)
```

Running this edition prints

```
MODEL: PS scheduling model
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.30861  0.30861  0.66667  0.66667  0.46291  0.46291
Delay    Class2    0.11625  0.11625  0.21667  0.21667  0.53653  0.53653
Queue1   Class1     1.6914  0.46291   3.6538   3.6538  0.46291  0.46291
Queue1   Class2     1.8838  0.53653    3.511    3.511  0.53653  0.53653

MODEL: FCFS scheduling model
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.30861  0.30861  0.66667  0.66667  0.46291  0.46291
Delay    Class2    0.11625  0.11625  0.21667  0.21667  0.53653  0.53653
Queue1   Class1     1.6914  0.46291   3.6538   3.6538  0.46291  0.46291
Queue1   Class2     1.8838  0.53653    3.511    3.511  0.53653  0.53653

MODEL: LCFS-PR scheduling model
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.30861  0.30861  0.66667  0.66667  0.46291  0.46291
Delay    Class2    0.11625  0.11625  0.21667  0.21667  0.53653  0.53653
Queue1   Class1     1.6914  0.46291   3.6538   3.6538  0.46291  0.46291
Queue1   Class2     1.8838  0.53653    3.511    3.511  0.53653  0.53653
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

2.2.3 `cqn_lcfs_multiclass`

Multiclass LCFS and LCFS-PR scheduling, and the difference preemption makes.

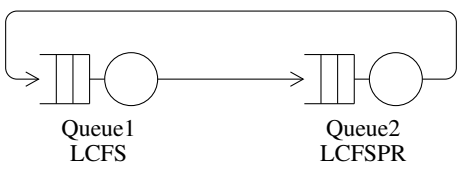

Figure 2.10: Topology of `cqn_lcfs_multiclass`: a closed network over 2 queueing stations.

The example is built and solved as follows.

Nodes	2: Queue × 2
Classes	3: Class1 (closed, $N = 1$), Class2 (closed, $N = 1$), Class3 (closed, $N = 1$)
Scheduling	Queue1: LCFS; Queue2: LCFSPR
Service	Queue1 – Class1: Exp(1); Class2: Exp(0.3333); Class3: Exp(0.2) Queue2 – Class1: Exp(0.5); Class2: Exp(0.25); Class3: Exp(0.1667)
Solvers	MVA, CTMC

Table 2.10: Features of `cqn_lcfs_multiclass`.

```
G. Casale, "A family of multiclass LCFS queueing networks with
order-dependent product-form solutions", QUESTA 2026.
# Service rates: mu(i,r) = service rate at station i for class r
mu = 1./np.array([[1, 3, 5],    # Station 1 (Queue1)
                  [2, 4, 6]])   # Station 2 (Queue2)
R = mu.shape[1]  # number of classes
# Create the network model
model = Network('LCFS Multiclass Model')
# Create nodes
node = np.empty(2, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.LCFS)
node[1] = Queue(model, 'Queue2', SchedStrategy.LCFSPR)
# Create job classes (one job per class)
jobclass = np.empty(R, dtype=object)
for r in range(R):
    jobclass[r] = ClosedClass(model, f'Class{r+1}', 1, node[0], 0)
# Set service times (exponential distributions)
for r in range(R):
    node[0].set_service(jobclass[r], Exp(mu[0, r]))
    node[1].set_service(jobclass[r], Exp(mu[1, r]))
# Set up routing: jobs alternate between the two queues
P = model.init_routing_matrix()
for r in range(R):
    P[jobclass[r], jobclass[r]] = np.array([[0, 1], [1, 0]])  # Queue1 -> Queue2 -> Queue1
model.link(P)
return model

    ... (15 source lines omitted; full implementation: python/examples/basic/)

avg_table_ctmc = solver_ctmc.avg_table()
print(avg_table_ctmc)

# Compare results
print('\n=== Comparison ===')
print('MVA and CTMC results should match for this product-form network.')
```

Running this edition prints

```
Solving LCFS+LCFS-PR network with MVA...

SOLVER: MVA
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT  ResidT      ArvR      Tput
Queue1   Class1    0.46099  0.20432  2.2562  2.2562   0.20432   0.20432
Queue1   Class2    0.31239  0.18744       5       5  0.062479  0.062479
Queue1   Class3     0.2381  0.17055  6.9802  6.9802   0.03411   0.03411
Queue2   Class1    0.53901  0.40865   2.638   2.638   0.20432   0.20432
Queue2   Class2    0.68761  0.24992  11.005  11.005  0.062479  0.062479
Queue2   Class3     0.7619  0.20466  22.337  22.337   0.03411   0.03411

Solving with CTMC for validation...

SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state

... (3 captured-output lines omitted; rerun the source example for the full output)

Queue1   Class2    0.31239  0.18744       5       5  0.062479  0.062479
Queue1   Class3     0.2381  0.17055  6.9802  6.9802   0.03411   0.03411
Queue2   Class1    0.53901  0.40865   2.638   2.638   0.20432   0.20432
```

```
Queue2   Class2    0.68761  0.24992  11.005  11.005  0.062479  0.062479
Queue2   Class3     0.7619  0.20466  22.337  22.337   0.03411   0.03411

=== Comparison ===
MVA and CTMC results should match for this product-form network.
```

The rows repeat for each of the 2 solvers the script runs (MVA, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.4 `cqn_mmpp2_service`

MMPP(2) service times, a correlated service process that no product-form solver can represent exactly.

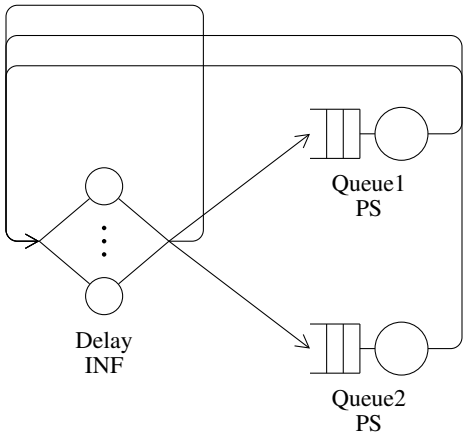

Figure 2.11: Topology of `cqn_mmpp2_service`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – Class1: Erlang(3,2); Class2: HyperExp(0.5,0.5;3,10) Queue1 – Class1: HyperExp(0.1,0.9;1,10); Class2: MMPP2 Queue2 – Class1: HyperExp(0.1,0.9;1,10); Class2: Erlang(1,2)
Features	class switching on 3 links
Solvers	JMT, LDES

Table 2.11: Features of `cqn_mmpp2_service`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Create network
model = Network('model')
# Block 1: nodes
node = np.empty(3, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
node[2] = Queue(model, 'Queue2', SchedStrategy.PS)
# Block 2: classes (1 job each)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 1, node[0], 0)
# %%
# Set service distributions
node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))
```

```
node[1].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[1].set_service(jobclass[1], MMPP2(1, 2, 3, 4))
node[2].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[2].set_service(jobclass[1], Erlang(1, 2))
# %%
# Set up connectivity
model.add_link(node[0], node[0])
model.add_link(node[0], node[1])

    ... (18 source lines omitted; full implementation: python/examples/basic/)

# %%
# Solve with each solver
for s in range(len(solver)):
    print(f'SOLVER: {solver[s].get_name()}')
    avg_table = solver[s].avg_table()
    print(avg_table)
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT    ResidT     ArvR     Tput
Delay    Class1     0.70972   0.70972   0.67253   0.67253   1.0537   1.0537
Delay    Class2     0.16657   0.16657   0.21263   0.21263  0.79236  0.79405
Queue1   Class1    0.066141  0.058043   0.21991  0.065973  0.32032  0.32032
Queue1   Class2     0.18831   0.18064   0.72928   0.24309  0.27172  0.27173
Queue2   Class1     0.22291   0.13915   0.29634   0.20744  0.74075  0.74564
Queue2   Class2     0.64052   0.55218    2.3056   0.76855  0.27048  0.27073
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station  JobClass      QLen      Util     RespT    ResidT     ArvR     Tput
Delay    Class1     0.70301   0.70212   0.66765   0.66765   1.0649   1.0532
Delay    Class2     0.17552   0.17648   0.21567   0.21567  0.82375  0.81454
Queue1   Class1    0.071872  0.059234   0.22614  0.067841  0.32236  0.31858
Queue1   Class2     0.20093    0.1883   0.73983   0.24661  0.27385  0.27128
Queue2   Class1     0.22512   0.14095   0.30647   0.21453  0.74258   0.7346
Queue2   Class2     0.62355   0.53938    2.3131   0.77104  0.27289  0.26965
```

The rows repeat for each of the 2 solvers the script runs (JMT, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.5 `cqn_multichain_cs`

Five classes in three chains, the closed counterpart of `oqn_multichain_cs`.

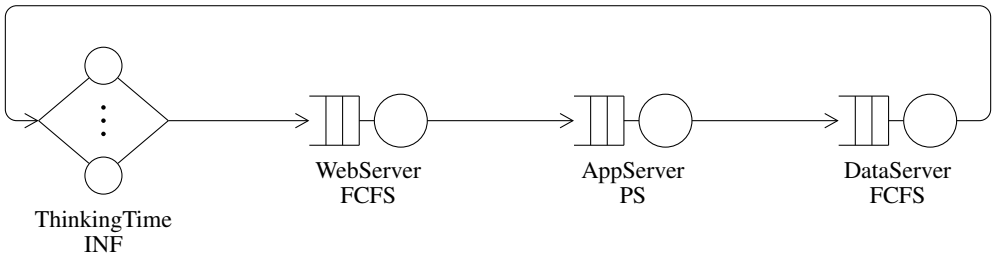

Figure 2.12: Topology of `cqn_multichain_cs`: a closed network over 4 queueing stations.

The example is built and solved as follows.

```
model = Network('cqn_multichain')
# Block 1: nodes
node = np.empty(4, dtype=object)
node[0] = Delay(model, 'ThinkingTime')
node[1] = Queue(model, 'WebServer', SchedStrategy.FCFS)
```

Nodes	4: Queue × 3, Delay
Classes	5: HighPriorityA (closed, $N = 2$), HighPriorityB (closed, $N = 1$), RegularA (closed, $N = 3$), RegularB (closed, $N = 2$), Background (closed, $N = 2$)
Scheduling	ThinkingTime: INF; WebServer: FCFS; AppServer: PS; DataServer: FCFS
Service	ThinkingTime – HighPriorityA: Exp(0.6667); HighPriorityB: Exp(0.625); RegularA: Exp(0.5); RegularB: Exp(0.4762); Background: Exp(0.3333) WebServer – HighPriorityA: Exp(2.5); HighPriorityB: Exp(2.381); RegularA: Exp(2); RegularB: Exp(1.923); Background: Exp(1.25) AppServer – HighPriorityA: Exp(1.429); HighPriorityB: Exp(1.333); RegularA: Exp(1.111); RegularB: Exp(1.053); Background: Exp(0.8333) DataServer – HighPriorityA: Exp(2); HighPriorityB: Exp(1.923); RegularA: Exp(1.667); RegularB: Exp(1.613); Background: Exp(1)
Solvers	MVA, SSA

Table 2.12: Features of `cqn_multichain_cs`.

```
node[2] = Queue(model, 'AppServer', SchedStrategy.PS)
node[3] = Queue(model, 'DataServer', SchedStrategy.FCFS)
# Block 2: job classes
# Chain 1: High-priority users (can switch between HighPriorityA and HighPriorityB)
jobclass = np.empty(5, dtype=object)
jobclass[0] = ClosedClass(model, 'HighPriorityA', 2, node[0])
jobclass[1] = ClosedClass(model, 'HighPriorityB', 1, node[0])
# Chain 2: Regular users (can switch between RegularA and RegularB)
jobclass[2] = ClosedClass(model, 'RegularA', 3, node[0])
jobclass[3] = ClosedClass(model, 'RegularB', 2, node[0])
# Chain 3: Background tasks
jobclass[4] = ClosedClass(model, 'Background', 2, node[0])
# Block 3: service times (all exponential for product-form)
# ThinkingTime (infinite server node)
node[0].set_service(jobclass[0], Exp.fit_mean(1.5))
node[0].set_service(jobclass[1], Exp.fit_mean(1.6))
node[0].set_service(jobclass[2], Exp.fit_mean(2.0))
node[0].set_service(jobclass[3], Exp.fit_mean(2.1))
node[0].set_service(jobclass[4], Exp.fit_mean(3.0))
# WebServer (FCFS, exponential)
node[1].set_service(jobclass[0], Exp.fit_mean(0.4))

    ... (61 source lines omitted; full implementation: python/examples/basic/)

solver = np.append(solver, SSA(model, seed=23000, samples=200000))

for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table = solver[s].avg_chain_table()
    print(avg_table)
```

Running this edition prints

```
SOLVER: MVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station       Chain      QLen     Util    RespT   ResidT     ArvR     Tput
ThinkingTime  Chain1  0.57374  0.57374   1.5545   1.5545  0.36907  0.36907
ThinkingTime  Chain2   1.0321   1.0321   2.0538   2.0538  0.50252  0.50252
ThinkingTime  Chain3   0.4589   0.4589        3        3  0.15297  0.15297
WebServer     Chain1  0.33996  0.15165  0.92113  0.92113  0.36907  0.36907
WebServer     Chain2  0.51027  0.25667   1.0154   1.0154  0.50252  0.50252
WebServer     Chain3  0.19558  0.12237   1.2786   1.2786  0.15297  0.15297
AppServer     Chain1   1.5456  0.26842   4.1879   4.1879  0.36907  0.36907
AppServer     Chain2   2.6732   0.4658   5.3196   5.3196  0.50252  0.50252
AppServer     Chain3   1.0548  0.18356   6.8954   6.8954  0.15297  0.15297
DataServer    Chain1  0.54066  0.18856   1.4649   1.4649  0.36907  0.36907
DataServer    Chain2  0.78445  0.30692    1.561    1.561  0.50252  0.50252
DataServer    Chain3  0.29075  0.15297   1.9007   1.9007  0.15297  0.15297


... (7 captured-output lines omitted; rerun the source example for the full output)

WebServer     Chain2  0.58458  0.26355  1.1332  1.1332  0.51588  0.51588
WebServer     Chain3  0.22276  0.13137  1.3565  1.3565  0.16421  0.16421
```

```
AppServer     Chain1    1.437  0.27269  3.8328  3.8328  0.37479  0.37479
AppServer     Chain2   2.4873  0.48448  4.7588  4.7588  0.52255  0.52255
AppServer     Chain3  0.97229  0.19251  6.0607  6.0607  0.16042  0.16042
DataServer    Chain1  0.59191  0.19148  1.5794  1.5794  0.37476  0.37476
DataServer    Chain2  0.87393  0.31503  1.6944  1.6944  0.51572  0.51572
DataServer    Chain3  0.32668  0.16361  1.9967  1.9967  0.16361  0.16361
```

The rows repeat for each of the 2 solvers the script runs (MVA, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.6 `cqn_multiserver`

A closed network with multiserver FCFS and PS stations, Erlang service and a class disabled at one station.

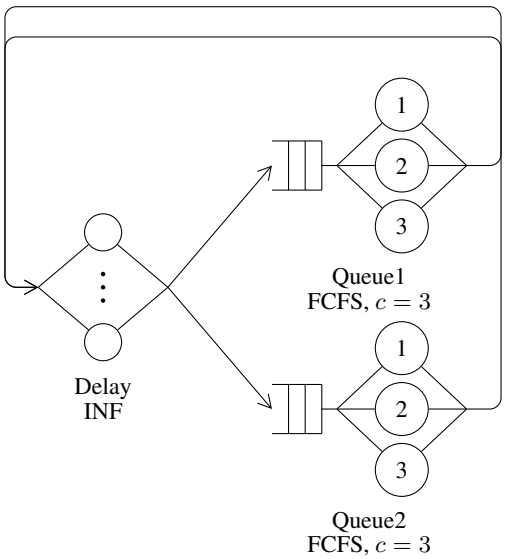

Figure 2.13: Topology of `cqn_multiserver`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	4: Class1 (closed, $N = 2$), Class2 (closed, $N = 2$), Class3 (closed, $N = 2$), Class4 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: FCFS ($c = 3$); Queue2: FCFS ($c = 3$)
Service	Delay – Class1: Exp(1); Class2: Exp(1); Class3: Exp(10); Class4: Exp(1) Queue1 – Class1: Exp(1); Class2: Erlang(1,2); Class3: Exp(10); Class4: Exp(1) Queue2 – Class1: Disabled; Class2: Disabled; Class3: Erlang(1,2); Class4: Exp(1)
Solvers	MVA

Table 2.13: Features of `cqn_multiserver`.

The example is built and solved as follows.

```
model = Network('model')
node = np.empty(3, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Queue(model, 'Queue2', SchedStrategy.FCFS)
node[1].set_number_of_servers(3)
node[2].set_number_of_servers(3)
jobclass = np.empty(4, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 0)
jobclass[2] = ClosedClass(model, 'Class3', 2, node[0], 0)
```

```
jobclass[3] = ClosedClass(model, 'Class4', 1, node[0], 0)
# Set service times
for r in range(4):
    if r < 3:
        node[0].set_service(jobclass[r], Exp(1) if r < 2 else Exp(10))
    else:
        node[0].set_service(jobclass[r], Exp(1))
node[1].set_service(jobclass[0], Exp(1))
node[1].set_service(jobclass[1], Erlang(1, 2))
node[1].set_service(jobclass[2], Exp(10))
node[1].set_service(jobclass[3], Exp(1))
node[2].set_service(jobclass[0], Disabled())
node[2].set_service(jobclass[1], Disabled())
node[2].set_service(jobclass[2], Erlang(1, 2))
node[2].set_service(jobclass[3], Exp(1))

    ... (48 source lines omitted; full implementation: python/examples/basic/)


# Run solver
print('\nSOLVER: MVA')
solver = MVA(model)
avg_table = solver.getAvgTable()
print(avg_table)
```

Running this edition prints

```
State (Running): [[4. 3. 2. 1. 0. 0. 0.]
 [4. 3. 2. 0. 1. 0. 0.]]
State (Started): [[4. 3. 2. 1. 0. 0. 0.]]
State (Marginal): [[4 3 2 1 0 0 0]
 [4 3 2 0 1 0 0]
 [4 2 2 0 0 1 0]
 [4 1 1 1 0 1 0]
 [4 1 1 0 1 1 0]
 [3 4 2 1 0 0 0]
 [3 4 2 0 1 0 0]
 [3 2 2 0 0 0 1]
 [3 1 1 1 0 0 1]
 [3 1 1 0 1 0 1]
 [2 4 2 0 0 1 0]
 [2 3 2 0 0 0 1]
 [2 1 1 0 0 1 1]]

... (12 captured-output lines omitted; rerun the source example for the full output)

Delay    Class2    0.47603  0.47603        1   0.33333  0.47603  0.47603
Delay    Class3    0.13232  0.13232      0.1  0.066667   1.3232   1.3232
Delay    Class4    0.66159  0.66159        1   0.33333  0.66159  0.66159
Queue1   Class1      1.286  0.31736   1.3507   0.90046  0.95207  0.95207
Queue1   Class2      1.286  0.31736   2.7014   0.90046  0.47603  0.47603
Queue1   Class3    0.19955  0.03308  0.20108   0.10054  0.99239  0.99239
Queue1   Class4     1.3303  0.22053   2.0108   0.67026  0.66159  0.66159
Queue2   Class3    0.67621  0.22053   2.0442    0.3407   0.3308   0.3308
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.2.7 `cqn_multiserver_nc`

Delay plus a multiserver FCFS queue, solved by the normalizing constant method.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: FCFS ($c = 3$)
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(1.25)
Solvers	CTMC, MVA, NC

Table 2.14: Features of `cqn_multiserver_nc`.

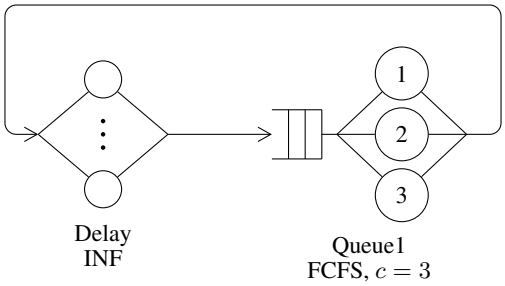

Figure 2.14: Topology of `cqn_multiserver_nc`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[1].set_number_of_servers(3)
jobclass = ClosedClass(model, 'Class1', 5, node[0], 0)
node[0].set_service(jobclass, Exp.fit_mean(1.0))  # mean = 1
node[1].set_service(jobclass, Exp.fit_mean(0.8))  # mean = 0.8
P = model.init_routing_matrix()
pmatrix = [[0, 1.0], [1.0, 0]]
for i in range(len(node)):
    for j in range(len(node)):
        P.set(jobclass, jobclass, node[i], node[j], pmatrix[i][j])
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)

model = cqn_multiserver_nc()

solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model))
solver = np.append(solver, MVA(model))
solver = np.append(solver, NC(model, method='exact'))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name().replace("Solver", "")}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1    2.6613   2.6613        1        1  2.6613  2.6613
Queue1   Class1    2.3387  0.70969  0.87875  0.87875  2.6613  2.6613

SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1    2.6613   2.6613        1        1  2.6613  2.6613
Queue1   Class1    2.3387  0.70969  0.87875  0.87875  2.6613  2.6613

SOLVER: NC
NC analysis [method: exact/gld; type: approximate, deterministic; lang: python; env: python] completed in
    <t>s.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1    2.6613   2.6613        1        1  2.6613  2.6613
Queue1   Class1    2.3387  0.70969  0.87875  0.87875  2.6613  2.6613
```

The rows repeat for each of the 3 solvers the script runs (CTMC, MVA, NC), which is the point of the

example: the same model read by methods of different cost and different exactness.

2.2.8 cqn_oneline

The closed counterpart of the one-line construction, from the demand and think time matrices.

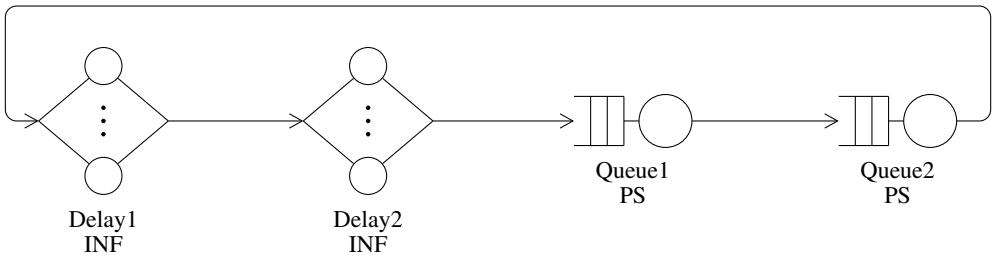

Figure 2.15: Topology of cqn_oneline: a closed network over 4 queueing stations.

Nodes	4: Queue × 2, Delay × 2
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 2$)
Scheduling	Delay1: INF; Delay2: INF; Queue1: PS; Queue2: PS
Service	Delay1 – Class1: Exp(0.01099); Class2: Exp(0.01087) Delay2 – Class1: Exp(0.01075); Class2: Exp(0.01064) Queue1 – Class1: Exp(0.1); Class2: Exp(0.2) Queue2 – Class1: Exp(0.2); Class2: Exp(0.1111)
Solvers	MVA

Table 2.15: Features of cqn_oneline.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Service times - S(i,r) - mean service time of class r at station i
D = np.array([[10, 5], [5, 9]])
# Number of jobs - N(r) - number of jobs of class r
N = np.array([1, 2])
# Think times - Z(r) - mean service time of class r at delay station i
Z = np.array([[91, 92], [93, 94]])
# %%
# Create cyclic model using the Network.cyclicPsInf method
model = Network.cyclicPsInf(N, D, Z)
# %%
# Solve using exact MVA
avgTable = MVA(model, method='exact').avg_table()
print(avgTable)
```

Running this edition prints

```
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass      QLen      Util    RespT  ResidT       ArvR       Tput
Delay1   Class1     0.45505   0.45505       91      91  0.0050005  0.0050005
Delay1   Class2     0.91525   0.91525       92      92  0.0099484  0.0099484
Delay2   Class1     0.46505   0.46505       93      93  0.0050005  0.0050005
Delay2   Class2     0.93515   0.93515       94      94  0.0099484  0.0099484
Queue1   Class1    0.052561  0.050005  10.511  10.511  0.0050005  0.0050005
Queue1   Class2    0.053601  0.049742  5.3879  5.3879  0.0099484  0.0099484
Queue2   Class1    0.027348  0.025002   5.469   5.469  0.0050005  0.0050005
Queue2   Class2    0.096001  0.089535  9.6499  9.6499  0.0099484  0.0099484
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by MVA.

2.2.9 cqn_repairmen

The machine repairman model: a Delay station of thinking machines and one FCFS repair queue.

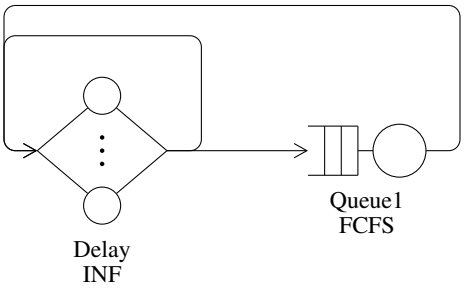

Figure 2.16: Topology of cqn_repairmen: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 10$)
Scheduling	Delay: INF; Queue1: FCFS
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(0.6667)
Solvers	CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES

Table 2.16: Features of cqn_repairmen.

The example is built and solved as follows.

```
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
jobclass = ClosedClass(model, 'Class1', 10, node[0], 0)
node[0].set_service(jobclass, Exp.fit_mean(1.0))  # mean = 1
node[1].set_service(jobclass, Exp.fit_mean(1.5))  # mean = 1.5
P = model.init_routing_matrix()
pmatrix = [[0.7, 0.3], [1.0, 0]]
for i in range(len(node)):
    for j in range(len(node)):
        P.set(jobclass, jobclass, node[i], node[j], pmatrix[i][j])
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)

model = cqn_repairmen()

# Run multiple solvers
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model))
solver = np.append(solver, JMT(model, seed=23000))
solver = np.append(solver, SSA(model, seed=23000, samples=5000))
solver = np.append(solver, FLD(model))
solver = np.append(solver, MVA(model))
solver = np.append(solver, NC(model, method='exact'))
solver = np.append(solver, MAM(model))
solver = np.append(solver, LDES(model, seed=23000, samples=5000))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
```

```
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass   QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    2.222    2.222       1       1    2.222    2.222
Queue1   Class1    7.778  0.99991  11.668  3.5004  0.66661  0.66661

SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    2.1909   2.1909  1.0081  1.0081   2.2124   2.2129
Queue1   Class1    7.7331  0.99977  11.573   3.472  0.67574  0.68022

SOLVER: SSA

... (26 captured-output lines omitted; rerun the source example for the full output)

Delay    Class1    2.222    2.222       1       1    2.222    2.222
Queue1   Class1    7.778  0.99991  11.668  3.5004  0.66661  0.66661

SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 5000 --seed 23000
Station  JobClass    QLen    Util    RespT   ResidT     ArvR     Tput
Delay    Class1    2.1647  2.1864  0.99139  0.99139   2.2025   2.1864
Queue1   Class1    7.8353       1   11.814   3.5441  0.66659  0.65971
```

The rows repeat for each of the 8 solvers the script runs (CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.10 `cqn_repairmen_multi`

The repairman model with several repair stations.

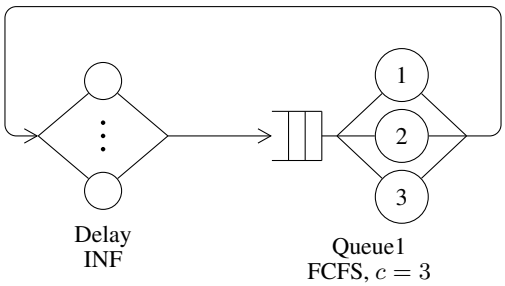

Figure 2.17: Topology of `cqn_repairmen_multi`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 4$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: FCFS ($c = 3$)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(1); Class2: Exp(10)
Solvers	CTMC, QNS, MVA, NC

Table 2.17: Features of `cqn_repairmen_multi`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
```

```
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[1].set_number_of_servers(3)
# Default: scheduling is set as FCFS everywhere, routing as Random
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 4, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 0)
node[0].set_service(jobclass[0], Exp(1))
node[0].set_service(jobclass[1], Exp(1))
node[1].set_service(jobclass[0], Exp(1))
node[1].set_service(jobclass[1], Exp(10))
myP = model.init_routing_matrix()
myP.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
myP.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
myP.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
myP.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(myP)
# %%
# Solve with multiple solvers
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model))

    ... (15 source lines omitted; full implementation: python/examples/basic/)

    try:
        AvgTable[s] = solver[s].avg_table()
        print(AvgTable[s])
    except Exception as e:
        print(f'  Error: {e}')
        AvgTable[s] = None
```

Running this edition prints

```
SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen      Util    RespT   ResidT    ArvR    Tput
Delay    Class1     1.9455    1.9455        1        1  1.9455  1.9455
Delay    Class2     1.6383    1.6383        1        1  1.6383  1.6383
Queue1   Class1     2.0545   0.64849   1.0561   1.0561  1.9455  1.9455
Queue1   Class2    0.36172  0.054609  0.22079  0.22079  1.6383  1.6383

SOLVER: QNS
LQNS command: lqns -Pmultiserver=conway -Pstop-on-message-loss=false -x <tmp>
Station  JobClass     QLen      Util    RespT   ResidT    ArvR    Tput
Delay    Class1     1.8692    1.8692        1        1  1.8692  1.8692
Delay    Class2      1.613     1.613        1        1   1.613   1.613
Queue1   Class1     2.1308   0.62307   1.1399   1.1399  1.8692  1.8692
Queue1   Class2    0.38701  0.053766  0.23994  0.23994   1.613   1.613

... (40 captured-output lines omitted; rerun the source example for the full output)


SOLVER: NC
NC analysis [method: comom; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen      Util    RespT   ResidT    ArvR    Tput
Delay    Class1     1.7852    1.7852        1        1  1.7852  1.7852
Delay    Class2     1.7418    1.7418        1        1  1.7418  1.7418
Queue1   Class1     2.2148   0.58399   1.2406   1.2406  1.7852  1.7852
Queue1   Class2     0.2582  0.063355  0.14824  0.14824  1.7418  1.7418
```

The rows repeat for each of the 4 solvers the script runs (CTMC, QNS, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.11 `cqn_scheduling_dps`

Discriminatory processor sharing, with the class weights that PS ignores.
The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
```

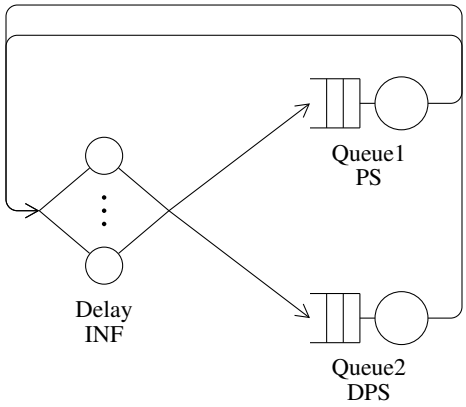

Figure 2.18: Topology of `cqn_scheduling_dps`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS; Queue2: DPS
Service	Delay – Class1: Exp(3); Class2: Exp(0.5) Queue1 – Class1: Exp(0.1); Class2: Exp(1) Queue2 – Class1: Exp(0.1); Class2: Exp(1)
Solvers	CTMC, JMT, FLD, MVA, LDES

Table 2.18: Features of `cqn_scheduling_dps`.

```
# %%
# Create network
model = Network('model')
# Block 1: nodes with mixed scheduling
node1 = Delay(model, 'Delay')
node2 = Queue(model, 'Queue1', SchedStrategy.PS)   # Processor Sharing
node3 = Queue(model, 'Queue2', SchedStrategy.DPS)  # Discriminatory Processor Sharing
# Block 2: classes with different populations
jobclass1 = ClosedClass(model, 'Class1', 2, node1, 0)  # 2 jobs
jobclass2 = ClosedClass(model, 'Class2', 1, node1, 0)  # 1 job
print("Network created with mixed scheduling:")
print(f"  {node1.get_name()}: Delay node")
print(f"  {node2.get_name()}: PS (Processor Sharing)")
print(f"  {node3.get_name()}: DPS (Discriminatory Processor Sharing)")
print(f"\nClass populations: Class1={jobclass1.get_population()}, Class2={jobclass2.
    get_population()}")
# %%
# Set service distributions
node1.set_service(jobclass1, Exp.fit_mean(1.0/3.0))   # Delay, Class1: mean=1/3
node1.set_service(jobclass2, Exp.fit_mean(1.0/0.5))   # Delay, Class2: mean=1/0.5=2
# PS Queue - weights are ignored in PS scheduling
w1_ps = 5  # This weight is ignored since the node is PS
w2_ps = 1  # This weight is ignored since the node is PS
node2.set_service(jobclass1, Exp.fit_mean(1.0/0.1), w1_ps)
node2.set_service(jobclass2, Exp.fit_mean(1.0/1.0), w2_ps)

    ... (49 source lines omitted; full implementation: python/examples/basic/)

        avg_table = solver.avg_table()
        avg_tables.append(avg_table)
        print(avg_table)
    except RuntimeError as e:
        print(f"Solver not supported for this model: {e}")
        avg_tables.append(None)
```

Running this edition prints

```
Network created with mixed scheduling:
  Delay: Delay node
```

```
  Queue1: PS (Processor Sharing)
  Queue2: DPS (Discriminatory Processor Sharing)

Class populations: Class1=2, Class2=1
Service parameters configured:
  Delay: Class1=Exp(3), Class2=Exp(0.5)
  Queue1 (PS): Class1=Exp(0.1), Class2=Exp(1) - weights ignored
  Queue2 (DPS): Class1=Exp(0.1,w=1), Class2=Exp(1,w=5) - Class2 has priority
Routing configured:
  Class1: 30% Queue1, 70% Queue2 (more traffic to DPS queue)
  Class2: 70% Queue1, 30% Queue2 (more traffic to PS queue)
  All queues return 100% to Delay
=== Multi-Solver Analysis ===
Configured 5 solvers for comparison

... (45 captured-output lines omitted; rerun the source example for the full output)

SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23000
Station  JobClass      QLen      Util     RespT    ResidT      ArvR      Tput
Delay    Class1    0.036266  0.035588  0.34036  0.34036   0.10789   0.10677
Delay    Class2     0.58336    0.5862   1.9881   1.9881   0.29455    0.2931
Queue1   Class1      0.6073    0.3676   17.458   5.2374  0.034998  0.034596
Queue1   Class2     0.29792   0.19935   1.4621   1.0235   0.20508   0.20404
Queue2   Class1      1.3564   0.74353   18.694   13.086  0.072893  0.072169
Queue2   Class2     0.11871  0.091908   1.3336  0.40007  0.089387  0.089065
```

The rows repeat for each of the 5 solvers the script runs (CTMC, JMT, FLD, MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.12 `cqn_threeclass_hyperl`

The three-class version, with multiserver stations.

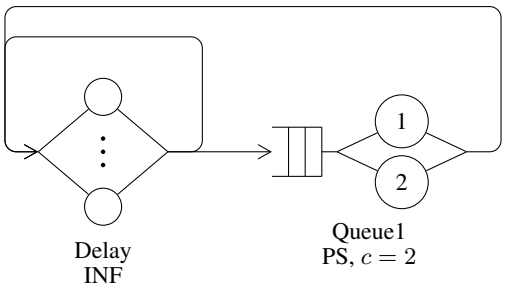

Figure 2.19: Topology of `cqn_threeclass_hyperl`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	3: Class1 (closed, $N = 2$), Class2 (closed, $N = 0$), Class3 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS ($c = 2$)
Service	Delay – Class1: Erlang(3,2); Class2: HyperExp(0.5,0.5;3,10); Class3: Exp(1) Queue1 – Class1: HyperExp(0.1,0.9;1,10); Class2: Exp(2); Class3: Exp(3)
Solvers	CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES

Table 2.19: Features of `cqn_threeclass_hyperl`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# %%
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
node[1].set_number_of_servers(2)
jobclass = np.empty(3, dtype=object)
```

```
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 0, node[0], 0)
jobclass[2] = ClosedClass(model, 'Class3', 1, node[0], 0)
node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))
node[0].set_service(jobclass[2], Exp(1))
node[1].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[1].set_service(jobclass[1], Exp(2))
node[1].set_service(jobclass[2], Exp(3))
P = model.init_routing_matrix()
# P{1,1} = [0.3,0.1; 0.2,0]
P.set(jobclass[0], jobclass[0], node[0], node[0], 0.3)
P.set(jobclass[0], jobclass[0], node[0], node[1], 0.1)
P.set(jobclass[0], jobclass[0], node[1], node[0], 0.2)
# P{1,2} = [0.6,0; 0.8,0]
P.set(jobclass[0], jobclass[1], node[0], node[0], 0.6)
P.set(jobclass[0], jobclass[1], node[1], node[0], 0.8)

    ... (29 source lines omitted; full implementation: python/examples/basic/)

    except Exception as e:
        print(f'Error with {solver[s].get_name()}: {str(e)[:100]}')
        # Set to None for failed solvers
        AvgTable[s] = None
        AvgChainTable[s] = None
        AvgSysTable[s] = None
```

Running this edition prints

```
SOLVER: CTMC
Station  JobClass      QLen     Util    RespT    ResidT     ArvR     Tput
Delay    Class1      1.1235   1.1235  0.66667   0.39683   1.6853   1.6853
Delay    Class2      0.2483   0.2483  0.21667  0.087698    1.146    1.146
Delay    Class3     0.74132  0.74132        1         1  0.74132  0.74132
Queue1   Class1    0.033246  0.01601  0.19727  0.011742  0.16853  0.16853
Queue1   Class2     0.59492   0.2865  0.51913   0.21012    1.146    1.146
Queue1   Class3     0.25868  0.12355  0.34894   0.34894  0.74132  0.74132

SOLVER: JMT
JMT Model: <tmp>
Station  JobClass      QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1      1.1301    1.1301  0.66008    0.3929   1.7071   1.7092
Delay    Class2     0.24356   0.24356  0.20999  0.084996   1.1618   1.1618
Delay    Class3     0.73801   0.73801   1.0024    1.0024  0.74033  0.74028
Queue1   Class1    0.038038  0.014299  0.20157  0.011999  0.16551  0.16612

... (59 captured-output lines omitted; rerun the source example for the full output)

SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 5000 --seed 23000
Station  JobClass     QLen      Util    RespT    ResidT    ArvR     Tput
Delay    Class1     1.0972    1.1009  0.66407   0.39528  1.6535   1.6513
Delay    Class2    0.25227   0.24391   0.2239  0.090626  1.1269   1.1258
Delay    Class3    0.73531     0.711   1.0327    1.0327  0.7121    0.711
Queue1   Class1    0.04344  0.021098  0.27517  0.016379   0.158   0.1569
Queue1   Class2    0.60704   0.29032  0.53909    0.2182  1.1269   1.1258
Queue1   Class3    0.26469   0.12544   0.3723    0.3723  0.7121  0.70991
```

The rows repeat for each of the 8 solvers the script runs (CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.13 `cqn_twoclass_erl`

Two classes with Erlang service over PS and INF stations, exercising the RAND, round-robin and weighted round-robin routing policies.
The example is built and solved as follows.

```
model = Network('model')
node = np.empty(4, dtype=object)
node[0] = ClassSwitch(model, 'CS', [[0, 1], [1, 0]])
```

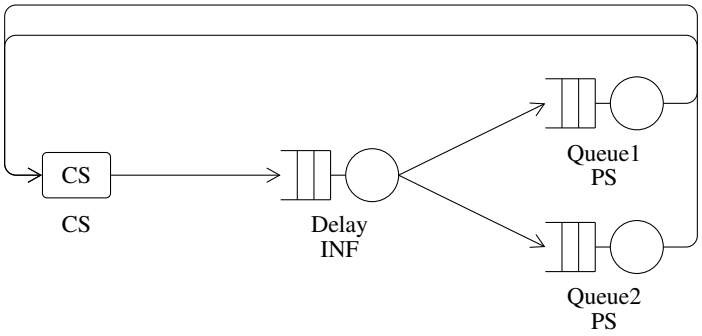

Figure 2.20: Topology of `cqn_twoclass_erl`: a closed network over 3 queueing stations with a class-switch node.

Nodes	4: Queue × 3, ClassSwitch
Classes	2: Class1 (closed, $N = 15$), Class2 (closed, $N = 5$)
Scheduling	Queue1: PS; Queue2: PS; Delay: INF
Service	Queue1 – Class1: Exp(0.6667); Class2: Erlang(1.333,2) Queue2 – Class1: Erlang(1.333,2); Class2: Exp(0.6667) Delay – Class1: Exp(1); Class2: Exp(1)
Features	class switching on 3 links
Solvers	JMT, LDES

Table 2.20: Features of `cqn_twoclass_erl`.

```
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
node[2] = Queue(model, 'Queue2', SchedStrategy.PS)
node[3] = Queue(model, 'Delay', SchedStrategy.INF)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 15, node[3], 0)
jobclass[1] = ClosedClass(model, 'Class2', 5, node[3], 0)
node[1].set_service(jobclass[0], Exp.fit_mean(1.5))  # mean = 1.5
node[1].set_service(jobclass[1], Erlang.fit_mean_and_order(1.5, 2))  # mean = 1.5
node[2].set_service(jobclass[0], Erlang.fit_mean_and_order(1.5, 2))  # mean = 1.5
node[2].set_service(jobclass[1], Exp.fit_mean(1.5))  # mean = 1.5
node[3].set_service(jobclass[0], Exp.fit_mean(1.0))  # mean = 1
node[3].set_service(jobclass[1], Exp.fit_mean(1.0))  # mean = 1
# Set routing
model.add_link(node[1], node[0])
model.add_link(node[2], node[0])
model.add_link(node[0], node[3])
model.add_link(node[3], node[1])
model.add_link(node[3], node[2])
# Set routing strategies
node[0].set_routing(jobclass[0], RoutingStrategy.RAND)
node[1].set_routing(jobclass[0], RoutingStrategy.RAND)
node[2].set_routing(jobclass[0], RoutingStrategy.RAND)
node[3].set_routing(jobclass[0], RoutingStrategy.RROBIN)

    ... (13 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
```

```
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   Class1    0.90422   0.4215   3.0843  0.77107  0.28506  0.28397
Queue1   Class2     0.5383  0.28473   2.8414  0.47356  0.19131  0.18984
Queue2   Class1     7.4591  0.44033   26.209   6.5522    0.287  0.28698
Queue2   Class2       9.92  0.55965   26.615   8.8718   0.3808  0.37815
Delay    Class1    0.57561  0.57561   1.0061  0.50304  0.56853  0.56405
Delay    Class2    0.54055  0.54055  0.97771  0.48885  0.56807  0.56532

SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station  JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Queue1   Class1    0.95227  0.43593  3.3375  0.83436  0.28805  0.28629
Queue1   Class2    0.58898  0.28755  3.0916  0.51527  0.19203  0.19086
Queue2   Class1     7.3685  0.42701  25.738   6.4344  0.28805   0.2863
Queue2   Class2     9.9434  0.57299  26.036   8.6786  0.38404   0.3817
Delay    Class1    0.57314  0.57258  1.0008  0.50038  0.57611  0.57258
Delay    Class2    0.57374  0.57258  1.0021  0.50104  0.57608  0.57258
```

The rows repeat for each of the 2 solvers the script runs (JMT, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.14 `cqn_twoclass_hyperl`

Two classes with hyperexponential and Erlang service on PS stations.

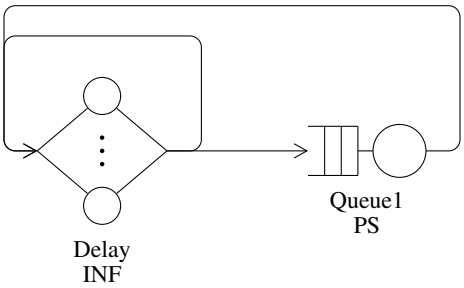

Figure 2.21: Topology of `cqn_twoclass_hyperl`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Erlang(3,2); Class2: HyperExp(0.5,0.5;3,10) Queue1 – Class1: HyperExp(0.1,0.9;1,10); Class2: Exp(1)
Solvers	CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES

Table 2.21: Features of `cqn_twoclass_hyperl`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 0)
node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))
node[1].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[1].set_service(jobclass[1], Exp(1))
P = model.init_routing_matrix()
P.set(jobclass[0], jobclass[0], node[0], node[0], 0.3)
```

```
P.set(jobclass[0], jobclass[0], node[0], node[1], 0.1)
P.set(jobclass[0], jobclass[0], node[1], node[0], 0.2)
P.set(jobclass[0], jobclass[1], node[0], node[0], 0.6)
P.set(jobclass[0], jobclass[1], node[1], node[0], 0.8)
P.set(jobclass[1], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
model.link(P)
# %%
# Aligned with JAR test scenarios for cqn_twoclass_hyperl
# JAR tests: CTMC(), JMT(seed=23000, samples=5000), SSA(seed=23000, samples=5000),

    ... (17 source lines omitted; full implementation: python/examples/basic/)

solver = np.append(solver, LDES(model, seed=23000, samples=5000))
AvgTable = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    AvgTable[s] = solver[s].avg_table()
    print(AvgTable[s])
```

Running this edition prints

```
SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass      QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1     0.93078   0.93078  0.66667   0.39683   1.3962   1.3962
Delay    Class2      0.2057    0.2057  0.21667  0.087698  0.94939  0.94939
Queue1   Class1    0.077835  0.026527  0.55749  0.033184  0.13962  0.13962
Queue1   Class2      2.7857   0.94939   2.9342    1.1876  0.94939  0.94939

SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass      QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1     0.93008   0.93008  0.66993   0.39877   1.4084   1.4078
Delay    Class2     0.20202   0.20202  0.21432  0.086747   0.9506  0.95064

... (42 captured-output lines omitted; rerun the source example for the full output)


SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 5000 --seed 23000
Station  JobClass      QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1     0.93661   0.95035   0.6577   0.39149    1.434   1.4255
Delay    Class2     0.21271   0.21055  0.21839  0.088398  0.97881  0.97176
Queue1   Class1    0.097085  0.034099  0.69505  0.041372  0.13973  0.13902
Queue1   Class2      2.7536   0.94186   2.8126    1.1384  0.97881  0.97105
```

The rows repeat for each of the 8 solvers the script runs (CTMC, JMT, SSA, FLD, MVA, NC, MAM, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.2.15 `cqn_twoqueues_multi`

Two FCFS queues and a Delay with several closed classes.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 10$), Class2 (closed, $N = 10$)
Scheduling	Delay: INF; Queue1: FCFS; Queue2: FCFS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.6667) Queue2 – Class1: Exp(0.3333); Class2: Exp(0.3333)
Solvers	MVA

Table 2.22: Features of `cqn_twoqueues_multi`.

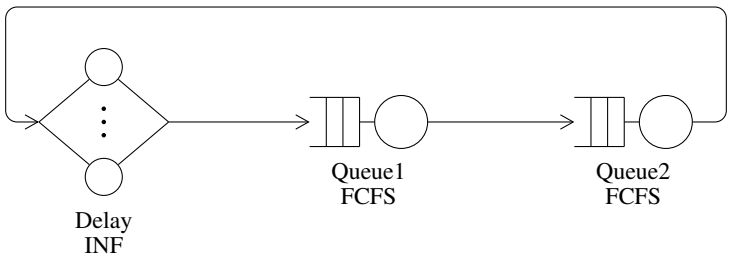

Figure 2.22: Topology of `cqn_twoqueues_multi`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
model = Network('model')
# Nodes
node = np.empty(3, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Queue(model, 'Queue2', SchedStrategy.FCFS)
# Job classes
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 10, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 10, node[0], 0)
# Service times
node[0].set_service(jobclass[0], Exp.fit_mean(1.0))   # mean = 1
node[1].set_service(jobclass[0], Exp.fit_mean(1.5))   # mean = 1.5
node[2].set_service(jobclass[0], Exp.fit_mean(3.0))   # mean = 3.0
node[0].set_service(jobclass[1], Exp.fit_mean(1.0))   # mean = 1
node[1].set_service(jobclass[1], Exp.fit_mean(1.5))   # mean = 1.5
node[2].set_service(jobclass[1], Exp.fit_mean(3.0))   # mean = 3.0
# Routing: serial routing for both classes
P = model.init_routing_matrix()
P[jobclass[0], jobclass[0]] = Network.serial_routing(node)
P[jobclass[1], jobclass[1]] = Network.serial_routing(node)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = cqn_twoqueues_multi()

# Solve with MVA
solver = MVA(model)
print(f'\nSOLVER: {solver.get_name()}')
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util  RespT  ResidT     ArvR     Tput
Delay    Class1    0.16667  0.16667      1       1  0.16667  0.16667
Delay    Class2    0.16667  0.16667      1       1  0.16667  0.16667
Queue1   Class1    0.49999     0.25      3       3  0.16667  0.16667
Queue1   Class2    0.49999     0.25      3       3  0.16667  0.16667
Queue2   Class1     9.3333      0.5     56      56  0.16667  0.16667
Queue2   Class2     9.3333      0.5     56      56  0.16667  0.16667
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.3 Mixed networks

A mixed network carries open and closed classes at the same stations. The open classes contribute a fixed arrival rate, the closed ones a fixed population, and the two compete for the same servers, so neither can be solved in isolation.

The scripts pair a single-server and a multiserver station under FCFS and under PS. They are deliberately awkward instances: the load-dependent rates they induce are sparse, and they are the models on which the approximate solvers disagree first.

2.3.1 `mqn_basic`

Open and closed classes at the same stations, the smallest mixed model.

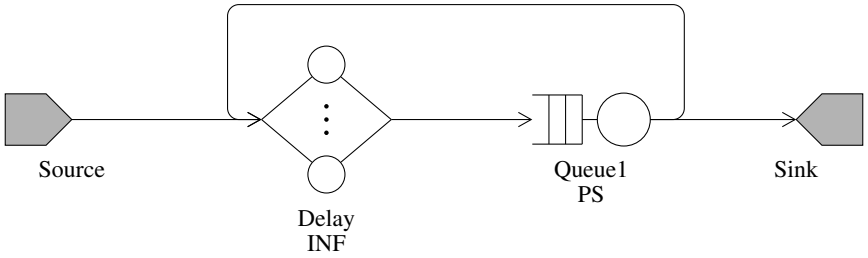

Figure 2.23: Topology of `mqn_basic`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	2: ClosedClass (closed, $N = 2$), OpenClass (open)
Scheduling	Delay: INF; Queue1: PS
Arrivals	Source – ClosedClass: Disabled; OpenClass: Exp(0.1)
Service	Delay – ClosedClass: Erlang(3,2); OpenClass: HyperExp(0.5,0.5;3,10) Queue1 – ClosedClass: HyperExp(0.1,0.9;1,10); OpenClass: Exp(1)
Features	class switching on 1 link
Solvers	CTMC, JMT, SSA, MVA, LDES

Table 2.23: Features of `mqn_basic`.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

node = np.empty(4, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')

jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'ClosedClass', 2, node[0], 0)
jobclass[1] = OpenClass(model, 'OpenClass', 0)

node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))

node[1].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[1].set_service(jobclass[1], Exp(1))

node[2].set_arrival(jobclass[1], Exp(0.1))

M = model.get_number_of_nodes()
```

```
K = model.get_number_of_classes()

P = model.init_routing_matrix()

    ... (19 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 3 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass         QLen      Util    RespT   ResidT      ArvR      Tput
Delay    ClosedClass    1.4362    1.4362  0.66667  0.66667    2.1543    2.1543
Delay    OpenClass    0.021607  0.021607  0.21667  0.21667  0.099723  0.099723
Queue1   ClosedClass    0.5638   0.40932  0.26171  0.26171    2.1543    2.1543
Queue1   OpenClass     0.17223  0.099723   1.7271   1.7271  0.099723  0.099723
Source   OpenClass           0         0        0        0         0  0.099723

SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000

... (28 captured-output lines omitted; rerun the source example for the full output)

SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station  JobClass         QLen      Util    RespT   ResidT      ArvR      Tput
Delay    ClosedClass    1.4343    1.4311  0.66819  0.66819    2.1489    2.1467
Delay    OpenClass    0.020437  0.021129   0.2097   0.2097  0.097608  0.097518
Queue1   ClosedClass   0.56573   0.41184  0.26347  0.26347    2.1488    2.1467
Queue1   OpenClass     0.16866  0.097673   1.7304   1.7304  0.097608  0.097518
Source   ClosedClass         0         0        0        0         0       NaN
Source   OpenClass           0         0        0        0         0       0.1
```

The rows repeat for each of the 5 solvers the script runs (CTMC, JMT, SSA, MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.3.2 `mqn_multichain_cs`

A mixed network whose classes are grouped into several chains by class switching.
The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('mqn_multichain')

# Block 1: nodes
node = np.empty(6, dtype=object)
node[0] = Delay(model, 'ThinkingTime')
node[1] = Queue(model, 'WebServer', SchedStrategy.FCFS)
node[2] = Queue(model, 'AppServer', SchedStrategy.PS)
node[3] = Queue(model, 'DataServer', SchedStrategy.FCFS)
node[4] = Source(model, 'Source')
node[5] = Sink(model, 'Sink')

# Block 2: job classes
jobclass = np.empty(5, dtype=object)
# Chain 1: Closed classes (interactive users that can switch types)
jobclass[0] = ClosedClass(model, 'InteractiveA', 3, node[0])
```

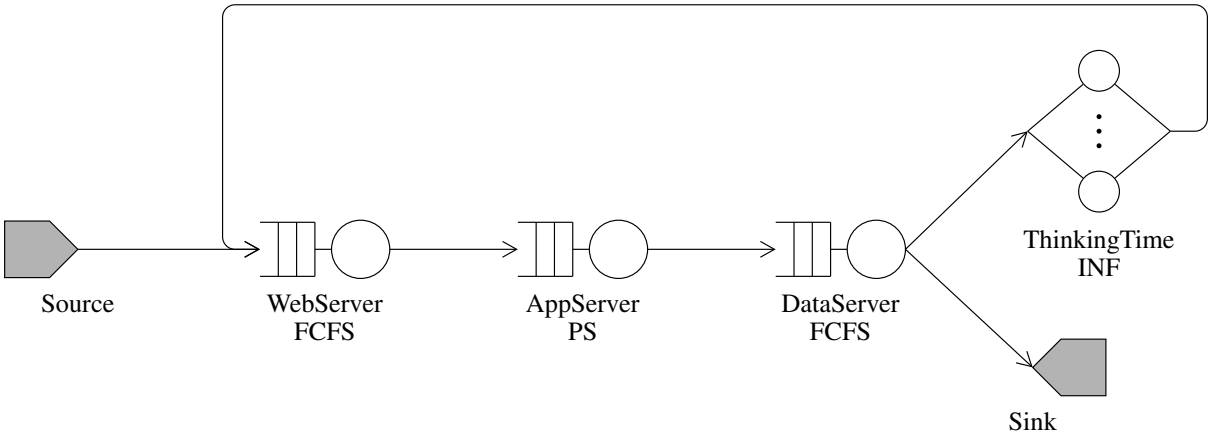

Figure 2.24: Topology of `mqn_multichain_cs`: an open network over 4 queueing stations.

Nodes	6: Source, Queue × 3, Delay, Sink
Classes	5: InteractiveA (closed, $N = 3$), InteractiveB (closed, $N = 2$), BatchA (open), BatchB (open), ExternalLoad (open)
Scheduling	ThinkingTime: INF; WebServer: FCFS; AppServer: PS; DataServer: FCFS
Arrivals	Source – InteractiveA: Disabled; InteractiveB: Disabled; BatchA: Exp(0.1); BatchB: Exp(0.05); ExternalLoad: Exp(0.05)
Service	ThinkingTime – InteractiveA: Exp(0.6667); InteractiveB: Exp(0.5); BatchA: Disabled; BatchB: Disabled; ExternalLoad: Disabled WebServer – InteractiveA: Exp(2.5); InteractiveB: Exp(2); BatchA: Exp(3.333); BatchB: Exp(2.857); ExternalLoad: Exp(5) AppServer – InteractiveA: Exp(1.25); InteractiveB: Exp(1); BatchA: Exp(1.667); BatchB: Exp(1.429); ExternalLoad: Exp(2) DataServer – InteractiveA: Exp(2); InteractiveB: Exp(1.667); BatchA: Exp(0.8333); BatchB: Exp(0.6667); ExternalLoad: Exp(1.25)
Features	drop rule {'BatchA': 'waitingQueue', 'BatchB': 'waitingQueue', 'ExternalLoad': 'waitingQueue'} at ThinkingTime; class switching on 1 link
Solvers	MVA, SSA

Table 2.24: Features of `mqn_multichain_cs`.

```
jobclass[1] = ClosedClass(model, 'InteractiveB', 2, node[0])
# Chain 2: Open classes (batch jobs that can switch types)
jobclass[2] = OpenClass(model, 'BatchA', 0)
jobclass[3] = OpenClass(model, 'BatchB', 0)
# Chain 3: Open class (external load)
jobclass[4] = OpenClass(model, 'ExternalLoad', 0)

# Block 3: service times
# ThinkingTime

    ... (78 source lines omitted; full implementation: python/examples/basic/)


for s in range(len(solver)):
    print(f'SOLVER: {solver[s].get_name().replace("Solver", "")}')
    avg_table = solver[s].avg_chain_table()
    print(avg_table)
    print()
```

Running this edition prints

```
Mixed QN with 3 chains, 5 classes, and class switching:
- Chain 1 (Closed): InteractiveA and InteractiveB (switch at WebServer)
- Chain 2 (Open): BatchA and BatchB (switch at WebServer)
- Chain 3 (Open): ExternalLoad

SOLVER: MVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station       Chain       QLen     Util    RespT   ResidT     ArvR     Tput
ThinkingTime  Chain1    1.2904   1.2904      1.8      1.8  0.71688  0.71688
ThinkingTime  Chain2         0        0        0        0        0        0
ThinkingTime  Chain3         0        0        0        0        0        0
WebServer     Chain1   0.47863  0.32977  0.66765  0.66765  0.71688  0.71688
WebServer     Chain2  0.085252   0.0475  0.56835  0.56835     0.15     0.15
WebServer     Chain3  0.022584     0.01  0.45168  0.45168     0.05     0.05
AppServer     Chain1    1.9763  0.65953   2.7568   2.7568  0.71688  0.71688
AppServer     Chain2    0.3307   0.0975   2.2047   2.2047     0.15     0.15

... (18 captured-output lines omitted; rerun the source example for the full output)

AppServer     Chain2    0.3242  0.096227   2.1915   2.1915   0.14736   0.14736
AppServer     Chain3  0.093886  0.027377   1.7147   1.7147  0.054754  0.054754
DataServer    Chain1    1.1535   0.43104   1.4964   1.4964   0.77219   0.77219
DataServer    Chain2   0.38426   0.19061    2.711    2.711   0.14243   0.14243
DataServer    Chain3  0.098618  0.041654    1.894    1.894  0.052068  0.052068
Source        Chain1         0         0        0        0         0         0
Source        Chain2         0         0   70.494   70.494      0.15      0.15
Source        Chain3         0         0     15.6     15.6      0.05      0.05
```

The rows repeat for each of the 2 solvers the script runs (MVA, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.3.3 `mqn_multiserver_fcfs`

The multiserver FCFS version of the same model.

Nodes	7: Source, Queue $\times$ 5, Sink
Classes	2: ClosedClass (closed, $N = 3$), OpenClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS ($c = 2$); Queue3: FCFS ($c = 3$); Queue4: FCFS ($c = 4$); Queue5: FCFS ($c = 5$)
Arrivals	Source – ClosedClass: Disabled; OpenClass: Exp(0.3)
Service	Queue1 – ClosedClass: Exp(1); OpenClass: Exp(1) Queue2 – ClosedClass: Exp(2); OpenClass: Exp(1.414) Queue3 – ClosedClass: Exp(3); OpenClass: Exp(1.732) Queue4 – ClosedClass: Exp(4); OpenClass: Exp(2) Queue5 – ClosedClass: Exp(5); OpenClass: Exp(2.236)
Features	class switching on 3 links
Solvers	CTMC, JMT, SSA, MVA, LDES

Table 2.25: Features of `mqn_multiserver_fcfs`.

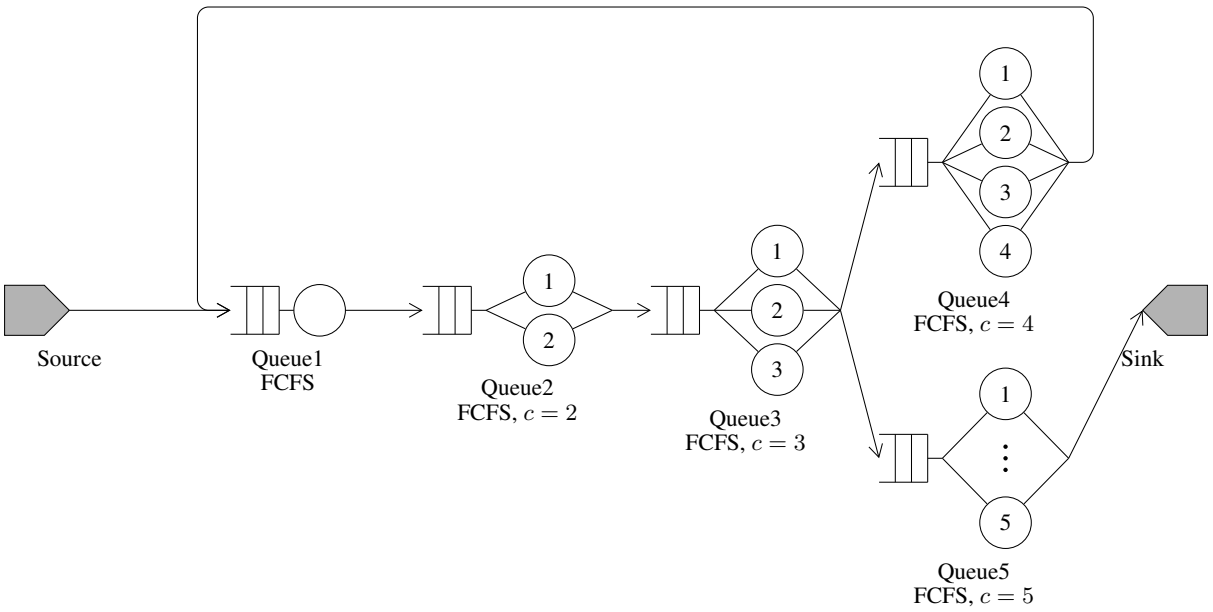

Figure 2.25: Topology of `mqn_multiserver_fcfs`: an open network over 5 queueing stations.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

node = np.empty(5, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[0].set_number_of_servers(1)
node[1] = Queue(model, 'Queue2', SchedStrategy.FCFS)
node[1].set_number_of_servers(2)
node[2] = Queue(model, 'Queue3', SchedStrategy.FCFS)
node[2].set_number_of_servers(3)
node[3] = Queue(model, 'Queue4', SchedStrategy.FCFS)  # only closed class
node[3].set_number_of_servers(4)
node[4] = Queue(model, 'Queue5', SchedStrategy.FCFS)  # only open class
node[4].set_number_of_servers(5)

source = Source(model, 'Source')
sink = Sink(model, 'Sink')

jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'ClosedClass', 3, node[0], 0)
jobclass[1] = OpenClass(model, 'OpenClass', 0)

for i in range(5):
    node[i].set_service(jobclass[0], Exp(i + 1))
    node[i].set_service(jobclass[1], Exp((i + 1) ** 0.5))

    ... (23 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
This example shows the execution of the solver on a 2-class mixed model with 5 multi-server FCFS nodes.
```

```
SOLVER: CTMC
CTMC solver using state space cutoff = 3 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass        QLen      Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass   2.1969      0.72   3.0513   3.0513     0.72     0.72
Queue1   OpenClass    0.95338   0.24763   3.8501   3.8501  0.24763  0.24763
Queue2   ClosedClass  0.38259      0.18  0.53137  0.53137     0.72     0.72
Queue2   OpenClass     0.1841  0.087549  0.74346  0.74346  0.24763  0.24763
Queue3   ClosedClass   0.2405      0.08  0.33402  0.33402     0.72     0.72
Queue3   OpenClass    0.14315  0.047656  0.57809  0.57809  0.24763  0.24763
Queue4   ClosedClass     0.18     0.045     0.25     0.25     0.72     0.72

... (50 captured-output lines omitted; rerun the source example for the full output)

Queue2   ClosedClass  0.36619   0.17175  0.53366  0.53366   0.6925  0.68555
Queue2   OpenClass    0.22113   0.10441  0.74333  0.74333  0.30126  0.29724
Queue3   ClosedClass  0.22512  0.074857  0.32834  0.32834  0.69246  0.68555
Queue3   OpenClass    0.17029  0.056675  0.57151  0.57151  0.30126  0.29724
Queue4   ClosedClass  0.17117  0.042793  0.24954  0.24954  0.69246  0.68555
Queue5   OpenClass    0.13333  0.026667   0.4485   0.4485  0.30126   0.2972
Source   ClosedClass        0         0        0        0        0      NaN
Source   OpenClass          0         0        0        0        0      0.3
```

The rows repeat for each of the 5 solvers the script runs (CTMC, JMT, SSA, MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.3.4 `mqn_multiserver_ps`

The multiserver PS version.

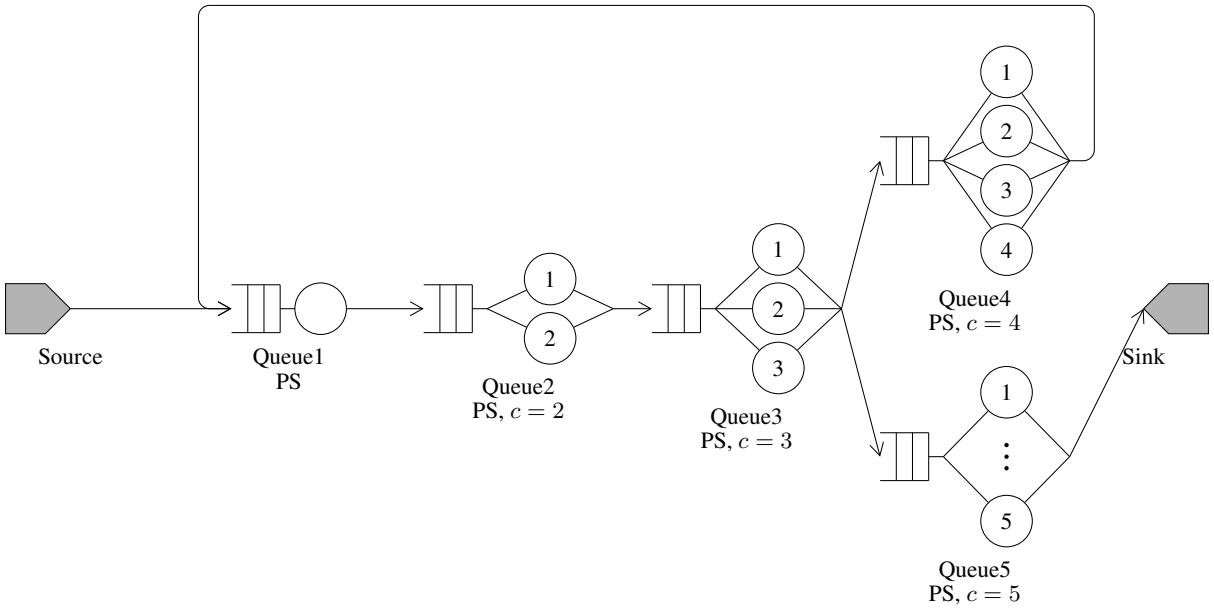

Figure 2.26: Topology of `mqn_multiserver_ps`: an open network over 5 queueing stations.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

node = np.empty(5, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.PS)
```

Nodes	7: Source, Queue × 5, Sink
Classes	2: ClosedClass (closed, $N = 3$), OpenClass (open)
Scheduling	Queue1: PS; Queue2: PS ($c = 2$); Queue3: PS ($c = 3$); Queue4: PS ($c = 4$); Queue5: PS ($c = 5$)
Arrivals	Source – ClosedClass: Disabled; OpenClass: Exp(0.3)
Service	Queue1 – ClosedClass: Exp(1); OpenClass: Exp(1) Queue2 – ClosedClass: Exp(2); OpenClass: Exp(1.414) Queue3 – ClosedClass: Exp(3); OpenClass: Exp(1.732) Queue4 – ClosedClass: Exp(4); OpenClass: Exp(2) Queue5 – ClosedClass: Exp(5); OpenClass: Exp(2.236)
Features	class switching on 3 links
Solvers	CTMC, JMT, MVA, LDES

Table 2.26: Features of `mqn_multiserver_ps`.

```
node[0].set_number_of_servers(1)
node[1] = Queue(model, 'Queue2', SchedStrategy.PS)
node[1].set_number_of_servers(2)
node[2] = Queue(model, 'Queue3', SchedStrategy.PS)
node[2].set_number_of_servers(3)
node[3] = Queue(model, 'Queue4', SchedStrategy.PS)  # only closed class
node[3].set_number_of_servers(4)
node[4] = Queue(model, 'Queue5', SchedStrategy.PS)  # only open class
node[4].set_number_of_servers(5)

source = Source(model, 'Source')
sink = Sink(model, 'Sink')

jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'ClosedClass', 3, node[0], 0)
jobclass[1] = OpenClass(model, 'OpenClass', 0)

for i in range(5):
    node[i].set_service(jobclass[0], Exp(i + 1))
    node[i].set_service(jobclass[1], Exp((i + 1) ** 0.5))

    ... (20 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 3 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass        QLen      Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass   2.1993   0.72064   3.0519   3.0519  0.72064  0.72064
Queue1   OpenClass     0.9534   0.24747   3.8526   3.8526  0.24747  0.24747
Queue2   ClosedClass  0.37998   0.18016  0.52728  0.52728  0.72064  0.72064
Queue2   OpenClass    0.18649  0.087494  0.75357  0.75357  0.24747  0.24747
Queue3   ClosedClass  0.24057  0.080071  0.33384  0.33384  0.72064  0.72064
Queue3   OpenClass    0.14316  0.047626  0.57849  0.57849  0.24747  0.24747
Queue4   ClosedClass  0.18016   0.04504     0.25     0.25  0.72064  0.72064
Queue5   OpenClass    0.11067  0.022134  0.44721  0.44721  0.24747  0.24747
Source   OpenClass          0         0        0        0        0  0.24747

... (34 captured-output lines omitted; rerun the source example for the full output)

Queue2   ClosedClass  0.36255   0.17132  0.52982  0.52982  0.68488  0.68417
Queue2   OpenClass    0.22359   0.10452  0.75073  0.75073  0.29805  0.29782
Queue3   ClosedClass   0.2245  0.074696  0.32813  0.32813  0.68488  0.68409
Queue3   OpenClass    0.17042  0.056662  0.57204  0.57204  0.29805  0.29782
Queue4   ClosedClass  0.17071  0.042677  0.24954  0.24954   0.6848  0.68409
```

```
Queue5   OpenClass    0.13359  0.026717  0.44842  0.44842  0.29805  0.29782
Source   ClosedClass        0         0        0        0        0      NaN
Source   OpenClass          0         0        0        0        0      0.3
```

The rows repeat for each of the 4 solvers the script runs (CTMC, JMT, MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.3.5 `mqn_singleserver_fcfs`

A single-server FCFS station under a mixed workload, a sparse load-dependent multiclass instance that stresses the solvers.

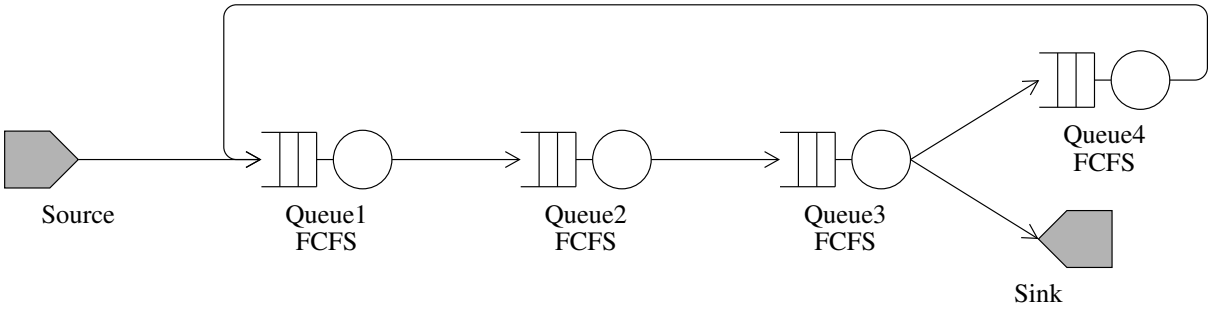

Figure 2.27: Topology of `mqn_singleserver_fcfs`: an open network over 4 queueing stations.

Nodes	6: Source, Queue × 4, Sink
Classes	2: ClosedClass (closed, $N = 100$), OpenClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS; Queue4: FCFS
Arrivals	Source – ClosedClass: Disabled; OpenClass: APH
Service	Queue1 – ClosedClass: Exp(1); OpenClass: Exp(1) Queue2 – ClosedClass: Exp(2); OpenClass: Exp(1.414) Queue3 – ClosedClass: Exp(3); OpenClass: Exp(1.732) Queue4 – ClosedClass: Exp(4); OpenClass: Exp(2)
Features	class switching on 2 links
Solvers	JMT, MVA, NC, LDES

Table 2.27: Features of `mqn_singleserver_fcfs`.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = Network('model')

M = 4
node = np.empty(M, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[1] = Queue(model, 'Queue2', SchedStrategy.FCFS)
node[2] = Queue(model, 'Queue3', SchedStrategy.FCFS)
node[3] = Queue(model, 'Queue4', SchedStrategy.FCFS)  # only closed class

source = Source(model, 'Source')
sink = Sink(model, 'Sink')

jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'ClosedClass', 100, node[0], 0)
jobclass[1] = OpenClass(model, 'OpenClass', 0)

for i in range(M):
    node[i].set_service(jobclass[0], Exp(i + 1))
    node[i].set_service(jobclass[1], Exp((i + 1) ** 0.5))
```

```
source.set_arrival(jobclass[1], APH.fit_mean_and_scv(3, 64))

# Routing
P = model.init_routing_matrix()

    ... (17 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass        QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass    98.44  0.68116   143.75   143.75  0.69274  0.68865
Queue1   OpenClass     56.624  0.29771   187.54   187.54  0.30425  0.30471
Queue2   ClosedClass  0.87067   0.3381   1.2547   1.2547  0.68865  0.68863
Queue2   OpenClass    0.53673  0.19108   1.7508   1.7508  0.30471  0.30472
Queue3   ClosedClass  0.43728  0.20633  0.62634  0.62634  0.68863  0.69273
Queue3   OpenClass    0.29725  0.18077  0.96959  0.96959  0.30472  0.29542
Queue4   ClosedClass  0.21893   0.1596  0.31419  0.31419  0.69273  0.69274
Source   OpenClass          0        0        0        0        0  0.30425

SOLVER: MVA
MVA analysis [method: lin; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (26 captured-output lines omitted; rerun the source example for the full output)

Queue1   OpenClass     59.101  0.32179   179.64   179.64  0.32901  0.32901
Queue2   ClosedClass  0.88249  0.35779   1.2535   1.2535  0.70461   0.7029
Queue2   OpenClass    0.57587  0.23679   1.7503   1.7503  0.32901  0.32901
Queue3   ClosedClass  0.43681  0.23124  0.62055  0.62055  0.70404  0.70328
Queue3   OpenClass    0.31057    0.185  0.94395  0.94395  0.32901  0.32901
Queue4   ClosedClass  0.23568  0.17988  0.33487  0.33487  0.70404  0.70309
Source   ClosedClass        0        0        0        0        0      NaN
Source   OpenClass          0        0        0        0        0  0.33333
```

The rows repeat for each of the 4 solvers the script runs (JMT, MVA, NC, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.3.6 `mqn_singleserver_ps`

The processor sharing counterpart.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
model = Network('model')
M = 4
node = np.empty(M, dtype=object)
node[0] = Queue(model, 'Queue1', SchedStrategy.PS)
node[1] = Queue(model, 'Queue2', SchedStrategy.PS)
node[2] = Queue(model, 'Queue3', SchedStrategy.PS)
node[3] = Queue(model, 'Queue4', SchedStrategy.PS)  # only closed class
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'ClosedClass', 100, node[0], 0)
jobclass[1] = OpenClass(model, 'OpenClass', 0)
for i in range(M):
    node[i].set_service(jobclass[0], Exp(i + 1))
```

```
    node[i].set_service(jobclass[1], Exp((i + 1) ** 0.5))
source.set_arrival(jobclass[1], Exp(0.3))
# Routing
P = model.init_routing_matrix()
# Closed class: serial routing through all 4 queues
P.set(jobclass[0], jobclass[0], Network.serial_routing(node[0], node[1], node[2], node[3]))
# Open class: source -> queue1 -> queue2 -> queue3 -> sink
P.set(jobclass[1], jobclass[1], Network.serial_routing(source, node[0], node[1], node[2],
    sink))
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)


    ... (14 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.4 Class switching

A job may change class as it moves. Switching is expressed either by off-diagonal blocks of the routing matrix, which give the probability of leaving node i as class r and arriving at node j as class s, or by a dedicated `ClassSwitch` node that performs the change in place.

Switching is what makes classes and chains differ: the classes that can reach one another form a chain, and it is the chain, not the class, that carries a conserved population. The diamond examples build a switching chain that is reducible on purpose, so that transient classes appear.

2.4.1 `cs_implicit`

A ClassSwitch node declared explicitly, against the implicit switching that a routing matrix with off-diagonal class blocks already expresses.

Figure 2.28: Topology of `cs_implicit`: an open network over 1 queueing station with a class-switch node.

Nodes	4: Source, Queue, ClassSwitch, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue 1: FCFS
Arrivals	Source 1 – Class1: Exp(0.1); Class2: Exp(0.5)
Service	Queue 1 – Class1: Exp(1); Class2: Exp(1)
Solvers	MVA

Table 2.28: Features of `cs_implicit`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Class switching example 1
# A basic M/M/1 with explicit definition of a ClassSwitch node
# Recommended ClassSwitch declaration style
model = Network('mm1cs')
# %%
# Block 1: nodes
node = np.empty(4, dtype=object)
node[0] = Source(model, 'Source 1')
node[1] = Queue(model, 'Queue 1', SchedStrategy.FCFS)
node[2] = Sink(model, 'Sink 1')
node[3] = ClassSwitch(model, 'ClassSwitch 1')
# Block 2: classes
jobclass = np.empty(2, dtype=object)
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
# %%
# Service configurations
node[0].set_arrival(jobclass[0], Exp.fit_mean(10.000000))  # (Source 1,Class1)
node[0].set_arrival(jobclass[1], Exp.fit_mean(2.000000))   # (Source 1,Class2)
node[1].set_service(jobclass[0], Exp.fit_mean(1.000000))   # (Queue 1,Class1)
node[1].set_service(jobclass[1], Exp.fit_mean(1.000000))   # (Queue 1,Class2)
# %%
# Block 3: topology

    ... (19 source lines omitted; full implementation: python/examples/basic/)

model.print_routing_matrix()
# %%
# Solve
solver = MVA(model)
AvgTable = solver.avg_chain_table()
print(AvgTable)
```

Running this edition prints

```
Source 1 [Class1] => ClassSwitch 1 [Class1] : Pr=1.000000
Source 1 [Class2] => ClassSwitch 1 [Class2] : Pr=1.000000
Queue 1 [Class1] => Sink 1 [Class1] : Pr=1.000000
Queue 1 [Class2] => Sink 1 [Class2] : Pr=1.000000
ClassSwitch 1 [Class1] => Queue 1 [Class1] : Pr=0.300000
ClassSwitch 1 [Class1] => Queue 1 [Class2] : Pr=0.700000
ClassSwitch 1 [Class2] => Queue 1 [Class1] : Pr=1.000000
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   Chain   QLen  Util  RespT  ResidT  ArvR  Tput
Source 1  Chain1     0     0      0       0     0   0.6
Queue 1   Chain1   1.5   0.6    2.5     2.5   0.6   0.6
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.4.2 `cs_multi_diamond`

The same pattern with several branches.
The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Class switching example 2
# Example of class switching controlled by a reducible Markov chain
model = Network('mm1cs')
# %%
# Block 1: nodes
node = np.empty(5, dtype=object)
node[0] = Source(model, 'Source 1')
```

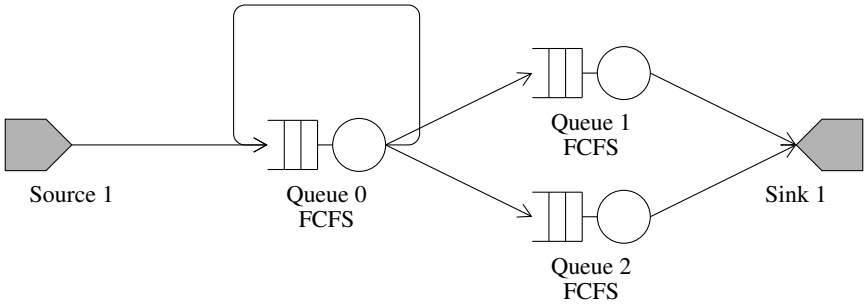

Figure 2.29: Topology of `cs_multi_diamond`: an open network over 3 queueing stations.

Nodes	5: Source, Queue × 3, Sink
Classes	3: Class1 (open), Class2 (open), Class3 (open)
Scheduling	Queue 0: FCFS; Queue 1: FCFS; Queue 2: FCFS
Arrivals	Source 1 – Class1: Exp(1); Class2: Disabled; Class3: Disabled
Service	Queue 0 – Class1: Exp(10); Class2: Disabled; Class3: Disabled Queue 1 – Class1: Disabled; Class2: Exp(20); Class3: Disabled Queue 2 – Class1: Disabled; Class2: Disabled; Class3: Exp(30)
Features	drop rule {'Class2': 'waitingQueue', 'Class3': 'waitingQueue'} at Queue 0; drop rule {'Class1': 'waitingQueue', 'Class3': 'waitingQueue'} at Queue 1; drop rule {'Class1': 'waitingQueue', 'Class2': 'waitingQueue'} at Queue 2; class switching on 1 link
Solvers	MVA

Table 2.29: Features of `cs_multi_diamond`.

```
node[1] = Queue(model, 'Queue 0', SchedStrategy.FCFS)
node[2] = Queue(model, 'Queue 1', SchedStrategy.FCFS)
node[3] = Queue(model, 'Queue 2', SchedStrategy.FCFS)
node[4] = Sink(model, 'Sink 1')
# Block 2: classes
jobclass = np.empty(3, dtype=object)
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
jobclass[2] = OpenClass(model, 'Class3', 0)
# %%
# Service configurations
# Note: Exp(rate) creates exponential with given rate (matching MATLAB's Exp(rate))
node[0].set_arrival(jobclass[0], Exp(1.0))    # (Source 1,Class1) rate=1
node[1].set_service(jobclass[0], Exp(10.0))   # (Queue 0,Class1) rate=10
node[2].set_service(jobclass[1], Exp(20.0))   # (Queue 1,Class2) rate=20
node[3].set_service(jobclass[2], Exp(30.0))   # (Queue 2,Class3) rate=30

    ... (10 source lines omitted; full implementation: python/examples/basic/)

model.print_routing_matrix()
# %%
# Solve
solver = MVA(model)
AvgTable = solver.avg_chain_table()
print(AvgTable)
```

Running this edition prints

```
Source 1 [Class1] => Queue 0 [Class1] : Pr=1.000000
Source 1 [Class2] => Queue 0 [Class2] : Pr=1.000000
Source 1 [Class3] => Queue 0 [Class3] : Pr=1.000000
Queue 0 [Class1] => Queue 0 [Class1] : Pr=0.200000
Queue 0 [Class1] => CS_Queue 0_to_Queue 1 [Class1] : Pr=0.300000
Queue 0 [Class1] => CS_Queue 0_to_Queue 2 [Class1] : Pr=0.500000
Queue 1 [Class2] => Sink 1 [Class2] : Pr=1.000000
Queue 2 [Class3] => Sink 1 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class1] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class2] => Queue 1 [Class2] : Pr=1.000000
```

```
CS_Queue 0_to_Queue 1 [Class3] => Queue 1 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class1] => Queue 2 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class2] => Queue 2 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class3] => Queue 2 [Class3] : Pr=1.000000
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   Chain       QLen      Util     RespT    ResidT   ArvR   Tput
Source 1  Chain1         0         0         0         0      0      1
Queue 0   Chain1   0.14286     0.125   0.11429   0.14286   1.25   1.25
Queue 1   Chain1  0.019108   0.01875  0.050955  0.019108  0.375  0.375
Queue 2   Chain1  0.021277  0.020833  0.034043  0.021277  0.625  0.625
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.4.3 `cs_single_diamond`

Class switching driven by a reducible Markov chain over the classes, the diamond pattern with one branch.

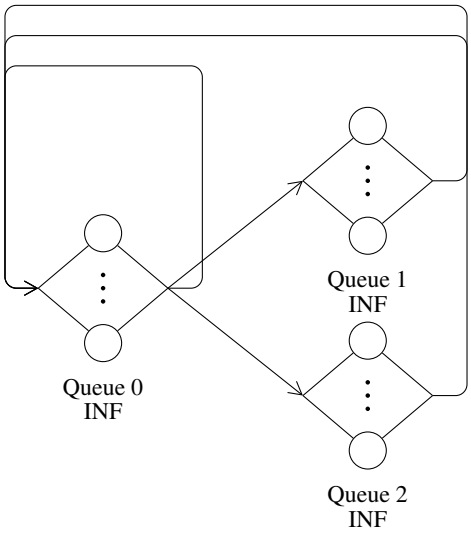

Figure 2.30: Topology of `cs_single_diamond`: a closed network over 3 queueing stations.

Nodes	3: Delay × 3
Classes	3: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 0$)
Scheduling	Queue 0: INF; Queue 1: INF; Queue 2: INF
Service	Queue 0 – Class1: Exp(1); Class2: Disabled; Class3: Disabled Queue 1 – Class1: Disabled; Class2: Exp(0.5); Class3: Disabled Queue 2 – Class1: Disabled; Class2: Disabled; Class3: Exp(0.3333)
Features	drop rule {'Class2': 'waitingQueue', 'Class3': 'waitingQueue'} at Queue 0; drop rule {'Class1': 'waitingQueue', 'Class3': 'waitingQueue'} at Queue 1; drop rule {'Class1': 'waitingQueue', 'Class2': 'waitingQueue'} at Queue 2
Solvers	MVA

Table 2.30: Features of `cs_single_diamond`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Class switching example 3
# Example of class switching controlled by a reducible Markov chain
```

```
model = Network('mm1cs')
# %%
# Block 1: nodes
node = np.empty(3, dtype=object)
node[0] = Delay(model, 'Queue 0')
node[1] = Delay(model, 'Queue 1')
node[2] = Delay(model, 'Queue 2')
# Block 2: classes
jobclass = np.empty(3, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0])
jobclass[1] = ClosedClass(model, 'Class2', 0, node[0])
jobclass[2] = ClosedClass(model, 'Class3', 0, node[0])
# %%
# Service configurations
node[0].set_service(jobclass[0], Exp.fit_mean(1.000000))  # (Queue 0,Class1)
node[1].set_service(jobclass[1], Exp.fit_mean(2.000000))  # (Queue 1,Class2)
node[2].set_service(jobclass[2], Exp.fit_mean(3.000000))  # (Queue 2,Class3)
# %%
# Routing matrix with class switching
P = model.init_routing_matrix()  # initialize routing matrix
P.set(jobclass[0], jobclass[0], node[0], node[0], 0.2)

    ... (5 source lines omitted; full implementation: python/examples/basic/)

model.print_routing_matrix()
# %%
# Solve
solver = MVA(model)
AvgTable = solver.avg_chain_table()
print(AvgTable)
```

Running this edition prints

```
Queue 0 [Class1] => Queue 0 [Class1] : Pr=0.200000
Queue 0 [Class1] => CS_Queue 0_to_Queue 1 [Class1] : Pr=0.300000
Queue 0 [Class1] => CS_Queue 0_to_Queue 2 [Class1] : Pr=0.500000
Queue 1 [Class2] => CS_Queue 1_to_Queue 0 [Class2] : Pr=1.000000
Queue 2 [Class3] => CS_Queue 2_to_Queue 0 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class1] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class2] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class3] => Queue 1 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class1] => Queue 2 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class2] => Queue 2 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class3] => Queue 2 [Class3] : Pr=1.000000
CS_Queue 1_to_Queue 0 [Class1] => Queue 0 [Class1] : Pr=1.000000
CS_Queue 1_to_Queue 0 [Class2] => Queue 0 [Class1] : Pr=1.000000
CS_Queue 1_to_Queue 0 [Class3] => Queue 0 [Class3] : Pr=1.000000
CS_Queue 2_to_Queue 0 [Class1] => Queue 0 [Class1] : Pr=1.000000
CS_Queue 2_to_Queue 0 [Class2] => Queue 0 [Class2] : Pr=1.000000
CS_Queue 2_to_Queue 0 [Class3] => Queue 0 [Class1] : Pr=1.000000
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  Chain      QLen     Util  RespT  ResidT      ArvR      Tput
Queue 0  Chain1  0.32258  0.32258      1       1   0.32258   0.32258
Queue 1  Chain1  0.19355  0.19355      2     0.6  0.096774  0.096774
Queue 2  Chain1  0.48387  0.48387      3     1.5   0.16129   0.16129
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.4.4 `cs_transient_class`

A class that is transient in the switching chain, so it is visited but never returned to.
The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Class switching example 4
# Example of class switching controlled by a reducible Markov chain
# In this variant the job remains either in class 2 or class 3 forever
```

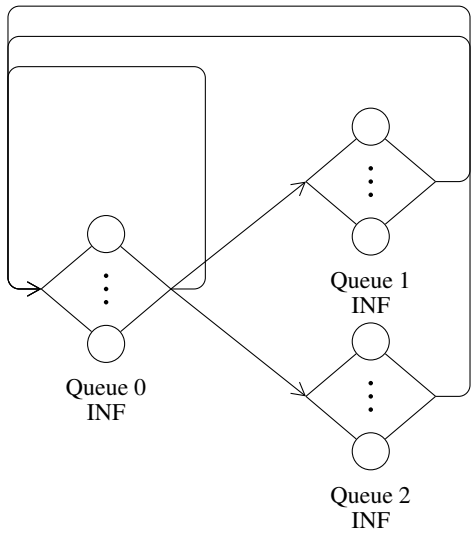

Figure 2.31: Topology of `cs_transient_class`: a closed network over 3 queueing stations.

Nodes	3: Delay × 3
Classes	3: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 0$)
Scheduling	Queue 0: INF; Queue 1: INF; Queue 2: INF
Service	Queue 0 – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1) Queue 1 – Class1: Disabled; Class2: Exp(1); Class3: Disabled Queue 2 – Class1: Disabled; Class2: Disabled; Class3: Exp(1)
Features	drop rule {'Class1': 'waitingQueue', 'Class3': 'waitingQueue'} at Queue 1; drop rule {'Class1': 'waitingQueue', 'Class2': 'waitingQueue'} at Queue 2
Solvers	MVA

Table 2.31: Features of `cs_transient_class`.

```
model = Network('reducible_cs')
# %%
# Block 1: nodes
node = np.empty(3, dtype=object)
node[0] = Delay(model, 'Queue 0')
node[1] = Delay(model, 'Queue 1')
node[2] = Delay(model, 'Queue 2')
# Block 2: classes
jobclass = np.empty(3, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0])
jobclass[1] = ClosedClass(model, 'Class2', 0, node[0])
jobclass[2] = ClosedClass(model, 'Class3', 0, node[0])
# %%
# Service configurations
node[0].set_service(jobclass[0], Exp.fit_mean(1.000000))  # (Queue 0,Class1)
node[0].set_service(jobclass[1], Exp.fit_mean(1.000000))  # (Queue 0,Class2)
node[0].set_service(jobclass[2], Exp.fit_mean(1.000000))  # (Queue 0,Class3)
node[1].set_service(jobclass[1], Exp.fit_mean(1.000000))  # (Queue 1,Class2)
node[2].set_service(jobclass[2], Exp.fit_mean(1.000000))  # (Queue 2,Class3)
# %%

    ... (15 source lines omitted; full implementation: python/examples/basic/)

model.print_routing_matrix()
# %%
# Solve
solver = MVA(model)
AvgTable = solver.avg_chain_table()
print(AvgTable)
```

Running this edition prints

```
Queue 0 [Class1] => Queue 0 [Class1] : Pr=0.200000
Queue 0 [Class1] => CS_Queue 0_to_Queue 1 [Class1] : Pr=0.300000
Queue 0 [Class1] => CS_Queue 0_to_Queue 2 [Class1] : Pr=0.500000
Queue 0 [Class2] => CS_Queue 0_to_Queue 1 [Class2] : Pr=1.000000
Queue 0 [Class3] => CS_Queue 0_to_Queue 2 [Class3] : Pr=1.000000
Queue 1 [Class2] => Queue 0 [Class2] : Pr=1.000000
Queue 2 [Class3] => Queue 0 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class1] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class2] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 1 [Class3] => Queue 1 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class1] => Queue 2 [Class3] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class2] => Queue 2 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class3] => Queue 2 [Class3] : Pr=1.000000
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  Chain     QLen    Util  RespT  ResidT    ArvR    Tput
Queue 0  Chain1     0.5     0.5      1       1     0.5     0.5
Queue 1  Chain1  0.1875  0.1875      1   0.375  0.1875  0.1875
Queue 2  Chain1  0.3125  0.3125      1   0.625  0.3125  0.3125
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.5 Priority models

Priorities are declared on the class and enforced by the discipline of the station. Head-of-line (HOL) service is non-preemptive: a low-priority job in service completes, and only the waiting order changes. The preemptive variants interrupt it instead, and the priority processor sharing family divides the capacity in proportion to the priority weights.

Priority breaks product form, so these models are the province of the exact CTMC solver, of simulation, and of the approximations that the MVA solver offers for HOL.

2.5.1 prio_hol_closed

The closed counterpart.

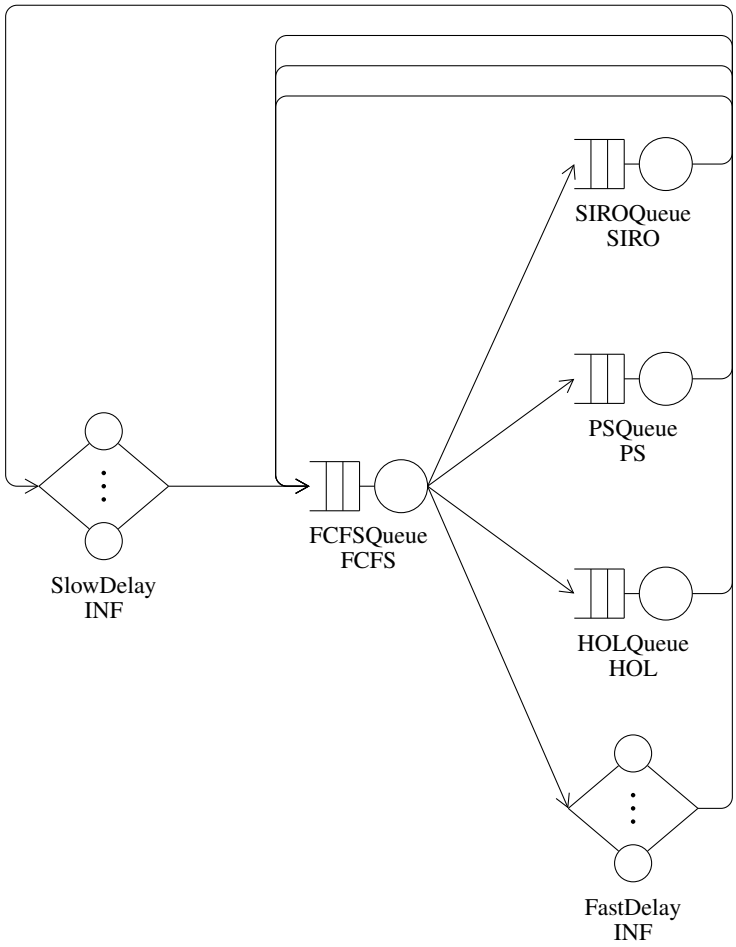

Figure 2.32: Topology of `prio_hol_closed`: a closed network over 6 queueing stations.

The example is built and solved as follows.

```
model = Network('MyNetwork')
# Nodes
node = np.empty(6, dtype=object)
node[0] = Delay(model, 'SlowDelay')
node[1] = Queue(model, 'FCFSQueue', SchedStrategy.FCFS)
node[2] = Queue(model, 'SIROQueue', SchedStrategy.SIRO)
node[3] = Queue(model, 'PSQueue', SchedStrategy.PS)
node[4] = Queue(model, 'HOLQueue', SchedStrategy.HOL)
node[5] = Delay(model, 'FastDelay')
# Classes (Class2 has priority 1, others have priority 0)
jobclass = np.empty(3, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 18, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 18, node[0], 1)
jobclass[2] = ClosedClass(model, 'Class3', 18, node[0], 0)
# Service times at slow delay
node[0].set_service(jobclass[0], Exp.fit_mean(10.0))
node[0].set_service(jobclass[1], Exp.fit_mean(10.0))
node[0].set_service(jobclass[2], Exp.fit_mean(10.0))
# Service times at FCFS queue
node[1].set_service(jobclass[0], Exp.fit_mean(0.3))
node[1].set_service(jobclass[1], Exp.fit_mean(0.5))
node[1].set_service(jobclass[2], Exp.fit_mean(0.6))
# Service times at SIRO queue
```

Nodes	6: Queue × 4, Delay × 2
Classes	3: Class1 (closed, $N = 18$), Class2 (closed, $N = 18$), Class3 (closed, $N = 18$)
Scheduling	SlowDelay: INF; FCFSQueue: FCFS; SIROQueue: SIRO; PSQueue: PS; HOLQueue: HOL; FastDelay: INF
Service	SlowDelay – Class1: Exp(0.1); Class2: Exp(0.1); Class3: Exp(0.1) FCFSQueue – Class1: Exp(3.333); Class2: Exp(2); Class3: Exp(1.667) SIROQueue – Class1: Exp(0.9091); Class2: Exp(0.7692); Class3: Exp(0.6667) PSQueue – Class1: Exp(1); Class2: Exp(0.9091); Class3: Exp(0.5263) HOLQueue – Class1: Exp(0.4); Class2: Exp(0.5263); Class3: Exp(0.2326) FastDelay – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1)
Solvers	MVA, JMT, SSA, LDES

Table 2.32: Features of `prio_hol_closed`.

```
node[2].set_service(jobclass[0], Exp.fit_mean(1.1))
node[2].set_service(jobclass[1], Exp.fit_mean(1.3))
node[2].set_service(jobclass[2], Exp.fit_mean(1.5))

    ... (55 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
Priority: highest=Class1,Class3, lowest=Class2
Priority: highest=Class1,Class3, lowest=Class2

SOLVER: MVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station    JobClass        QLen        Util   RespT  ResidT        ArvR        Tput
SlowDelay  Class1          1.4877      1.4877      10      10     0.14877     0.14877
SlowDelay  Class2       0.0078005   0.0078005      10      10  0.00078005  0.00078005
SlowDelay  Class3           1.447       1.447      10      10      0.1447      0.1447
FCFSQueue  Class1         0.49316     0.17853  0.8287  3.3148     0.59509     0.59509
FCFSQueue  Class2       0.0032351   0.0015601  1.0368  4.1473   0.0031202   0.0031202
FCFSQueue  Class3          0.6456     0.34728  1.1154  4.4616      0.5788      0.5788
SIROQueue  Class1         0.28096     0.16365  1.8885  1.8885     0.14877     0.14877
SIROQueue  Class2       0.0016424   0.0010141  2.1056  2.1056  0.00078005  0.00078005
SIROQueue  Class3         0.32966     0.21705  2.2782  2.2782      0.1447      0.1447
PSQueue    Class1         0.25406     0.14877  1.7077  1.7077     0.14877     0.14877

... (61 captured-output lines omitted; rerun the source example for the full output)

SIROQueue  Class1     0.30658  0.17016   2.0835   2.0835  0.21257  0.15104
SIROQueue  Class3     0.39882  0.23942   2.5953   2.5953  0.20437  0.15626
PSQueue    Class1     0.24624  0.14306   1.8929   1.8929  0.20325   0.1529
PSQueue    Class3     0.46392  0.26962   3.5796   3.5796  0.20064  0.14433
HOLQueue   Class1      15.059  0.36914   94.948   94.948  0.20213  0.14805
HOLQueue   Class3      14.807  0.63086   95.636   95.636  0.19728  0.14544
FastDelay  Class1     0.16083  0.15999  0.98504  0.98504  0.21257  0.15999
FastDelay  Class3     0.14722  0.14731  0.97941  0.97941  0.19616  0.14731
```

The rows repeat for each of the 4 solvers the script runs (MVA, JMT, SSA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.5.2 `prio_hol_open`

Head-of-line non-preemptive priority in an open network, with PS, SIRO and FCFS stations alongside. The example is built and solved as follows.

```
model = Network('MyNetwork')
# Nodes
node = np.empty(6, dtype=object)
```

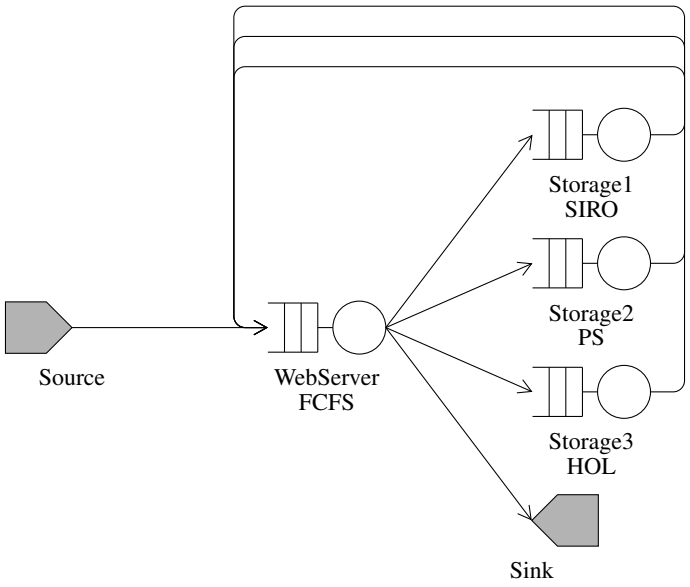

Figure 2.33: Topology of `prio_hol_open`: an open network over 4 queueing stations.

Nodes	6: Source, Queue × 4, Sink
Classes	3: Class1 (open), Class2 (open), Class3 (open)
Scheduling	WebServer: FCFS; Storage1: SIRO; Storage2: PS; Storage3: HOL
Arrivals	Source – Class1: Exp(0.1); Class2: Exp(0.1); Class3: Exp(0.1)
Service	WebServer – Class1: Exp(3.333); Class2: Exp(2); Class3: Exp(1.667) Storage1 – Class1: Exp(0.9091); Class2: Exp(0.7692); Class3: Exp(0.6667) Storage2 – Class1: Exp(0.5); Class2: Exp(0.4762); Class3: Exp(0.5263) Storage3 – Class1: Exp(0.4); Class2: Exp(0.5263); Class3: Exp(0.2326)
Solvers	CTMC, MVA, JMT, SSA

Table 2.33: Features of `prio_hol_open`.

```
node[0] = Source(model, 'Source')
node[1] = Queue(model, 'WebServer', SchedStrategy.FCFS)
node[2] = Queue(model, 'Storage1', SchedStrategy.SIRO)
node[3] = Queue(model, 'Storage2', SchedStrategy.PS)
node[4] = Queue(model, 'Storage3', SchedStrategy.HOL)
node[5] = Sink(model, 'Sink')
# Classes (Class2 has priority 1, others have priority 0)
jobclass = np.empty(3, dtype=object)
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 1)
jobclass[2] = OpenClass(model, 'Class3', 0)
# Arrivals
node[0].set_arrival(jobclass[0], Exp.fit_mean(10.0))
node[0].set_arrival(jobclass[1], Exp.fit_mean(10.0))
node[0].set_arrival(jobclass[2], Exp.fit_mean(10.0))
# Service times
node[1].set_service(jobclass[0], Exp.fit_mean(0.3))
node[1].set_service(jobclass[1], Exp.fit_mean(0.5))
node[1].set_service(jobclass[2], Exp.fit_mean(0.6))
node[2].set_service(jobclass[0], Exp.fit_mean(1.1))
node[2].set_service(jobclass[1], Exp.fit_mean(1.3))
node[2].set_service(jobclass[2], Exp.fit_mean(1.5))
node[3].set_service(jobclass[0], Exp.fit_mean(2.0))

    ... (48 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
Priority: highest=Class1,Class3, lowest=Class2
Priority: highest=Class1,Class3, lowest=Class2

SOLVER: CTMC
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station    JobClass      QLen      Util     RespT  ResidT      ArvR      Tput
Source     Class1           0         0         0       0         0  0.052281
Source     Class2           0         0         0       0         0  0.050449
Source     Class3           0         0         0       0         0  0.046082
WebServer  Class1    0.090117  0.062737  0.43092  1.7237   0.20912   0.20912
WebServer  Class2     0.12132    0.1009  0.60119  2.4048   0.20179   0.20179
WebServer  Class3     0.12559    0.1106  0.68136  2.7255   0.18433   0.18433

... (60 captured-output lines omitted; rerun the source example for the full output)

Storage1   Class2     0.26735   0.15575   2.2316  2.2316   0.10761   0.11981
Storage1   Class3     0.23201   0.14635    2.378   2.378   0.10307  0.097563
Storage2   Class1     0.51203   0.20712   4.9442  4.9442  0.099783   0.10356
Storage2   Class2     0.66933   0.25871   5.4331  5.4331   0.10761    0.1232
Storage2   Class3     0.42879   0.18429   4.4208  4.4208   0.10307  0.096992
Storage3   Class1      1.2392   0.26539   11.673  11.673  0.099783   0.10616
Storage3   Class2      8.5617   0.21107   77.071  77.071   0.10761   0.11109
Storage3   Class3      1.2076   0.38531   13.477  13.477   0.10307  0.089607
```

The rows repeat for each of the 4 solvers the script runs (CTMC, MVA, JMT, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.5.3 prio_identical

Priority classes with identical parameters, the sanity check that priority alone does not move the aggregate metrics.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
model = Network('model')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.GPSPRIO)
jobclass = np.empty(4, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 6, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 4, node[0], 1)
jobclass[2] = ClosedClass(model, 'Class3', 4, node[0], 1)
jobclass[3] = ClosedClass(model, 'Class4', 1, node[0], 2)
node[0].set_service(jobclass[0], Erlang(3.0, 2))
node[0].set_service(jobclass[1], Exp(1.0))
node[0].set_service(jobclass[2], Exp(1.0))
node[0].set_service(jobclass[3], Exp(2.0))
# Set service with weights for GPS
w1 = 12
node[1].set_service(jobclass[0], Exp(30), w1)
w2 = 3
node[1].set_service(jobclass[1], Exp(2), w2)
w3 = 5
node[1].set_service(jobclass[2], Exp(12), w3)
w4 = 1
node[1].set_service(jobclass[3], Exp(1), w4)
# Serial routing for all classes
P = model.init_routing_matrix()
for r in range(4):
    P[jobclass[r], jobclass[r]] = Network.serial_routing(node)

    ... (9 source lines omitted; full implementation: python/examples/basic/)


avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.5.4 prio_psprio

Priority processor sharing, where the share of a class is set by its priority.

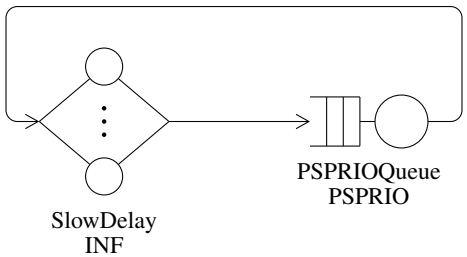

Figure 2.34: Topology of `prio_psprio`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 2$)
Scheduling	SlowDelay: INF; PSPRIOQueue: PSPRIO
Service	SlowDelay – Class1: Erlang(3,2); Class2: HyperExp(0.5,0.5;3,10) PSPRIOQueue – Class1: HyperExp(0.1,0.9;1,10); Class2: Exp(1)
Solvers	CTMC, JMT, SSA

Table 2.34: Features of `prio_psprio`.

The example is built and solved as follows.

```
model = Network('MyNetwork')
node = np.empty(2, dtype=object)
node[0] = Delay(model, 'SlowDelay')
node[1] = Queue(model, 'PSPRIOQueue', SchedStrategy.PSPRIO)
jobclass = np.empty(2, dtype=object)
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 1)
node[0].set_service(jobclass[0], Erlang(3, 2))
node[0].set_service(jobclass[1], HyperExp(0.5, 3.0, 10.0))
node[1].set_service(jobclass[0], HyperExp(0.1, 1.0, 10.0))
node[1].set_service(jobclass[1], Exp(1))
# Serial routing for both classes
P = model.init_routing_matrix()
P[jobclass[0], jobclass[0]] = Network.serial_routing(node)
P[jobclass[1], jobclass[1]] = Network.serial_routing(node)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = prio_psprio()

# Run solvers
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model))
solver = np.append(solver, JMT(model, seed=23000, verbose=True, samples=5000))
solver = np.append(solver, SSA(model, seed=23000, verbose=True, samples=5000))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
Priority: highest=Class1, lowest=Class2
Priority: highest=Class1, lowest=Class2

SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station      JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
SlowDelay    Class1     1.4834   1.4834  0.66667  0.66667   2.2252   2.2252
SlowDelay    Class2    0.12349  0.12349  0.21667  0.21667  0.56996  0.56996
PSPRIOQueue  Class1    0.51655  0.42278  0.23214  0.23214   2.2252   2.2252
PSPRIOQueue  Class2     1.8765  0.56996   3.2923   3.2923  0.56996  0.56996

SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.

... (5 captured-output lines omitted; rerun the source example for the full output)


SOLVER: SSA
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
Station      JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
SlowDelay    Class1     1.4506   1.4506  0.65585  0.65585   2.1514   2.2118
```

```
SlowDelay    Class2    0.13061  0.13061  0.21605  0.21605  0.55583  0.60456
PSPRIOQueue  Class1    0.54936  0.43671  0.25535  0.25535   2.2118   2.1514
PSPRIOQueue  Class2     1.8694  0.55583   3.3632   3.3632  0.60456  0.55583
```

The rows repeat for each of the 3 solvers the script runs (CTMC, JMT, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6 Fork-join networks

A `Fork` node splits an arriving job into sibling tasks, one per outgoing branch, and a `Join` node recombines them. The response time of the parent job is the maximum over its siblings, which is why fork-join has no product-form solution and why the join is where the modeling subtleties live: how many siblings are required (the quorum), whether the join is present at all, and what happens when branches share a station.

The family is the largest in the basic set because each of those questions has its own script. Nesting a fork inside a branch, three or more branches, asymmetric branch demands and class switching on the siblings are all covered separately, so a failure can be localized.

2.6.1 `fj_asymm`

Asymmetric branches, whose response time is set by the slowest one.

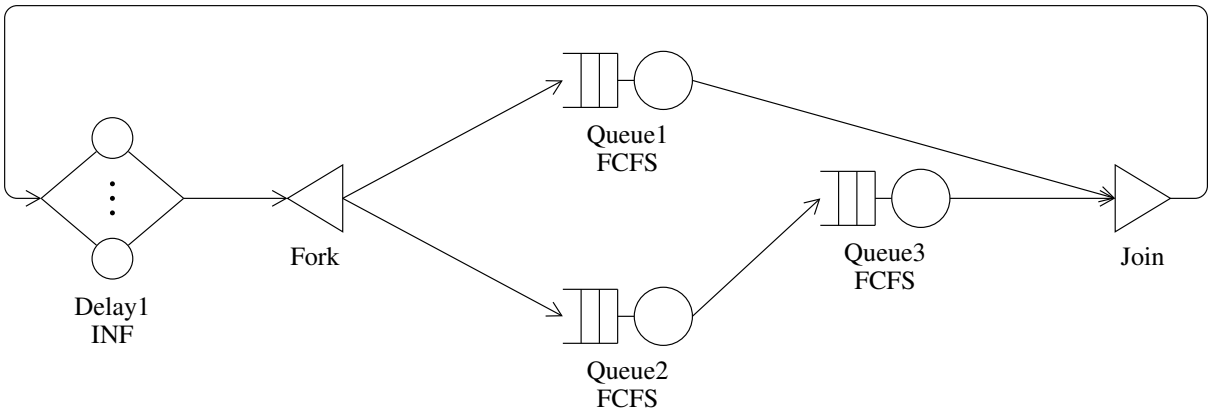

Figure 2.35: Topology of `fj_asymm`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue × 3, Delay, Fork, Join
Classes	1: class1 (closed, $N = 10$)
Scheduling	Delay1: INF; Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Delay1 – class1: Exp(0.5) Queue1 – class1: Exp(1) Queue2 – class1: Exp(2) Queue3 – class1: Exp(1)
Solvers	JMT, MVA

Table 2.35: Features of `fj_asymm`.

The example is built and solved as follows.

```
model = Network('model')
delay = Delay(model, 'Delay1')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
```

```
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(2.0))
queue3.set_service(jobclass1, Exp(1.0))
delay.set_service(jobclass1, Exp(0.5))
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, delay, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue1, join, 1.0)
P.set(jobclass1, jobclass1, queue2, queue3, 1.0)
P.set(jobclass1, jobclass1, queue3, join, 1.0)
P.set(jobclass1, jobclass1, join, delay, 1.0)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = fj_asymm()

solver = np.array([], dtype=object)
solver = np.append(solver, JMT(model, seed=23000))
solver = np.append(solver, MVA(model))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay1   class1     1.8108   1.8108  1.9614  1.9614  0.94914  0.94982
Queue1   class1     6.1513  0.95281  6.5892  6.5892  0.94982  0.94864
Queue2   class1    0.82403  0.47064  0.8557  0.8557  0.94982  0.95192
Queue3   class1     5.1269  0.93083  5.4049  5.4049  0.95192  0.94412
Join     class1      4.293        0  2.2396  2.2396   1.8954  0.94685

SOLVER: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (5 captured-output lines omitted; rerun the source example for the full output)

MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.7139   1.7139        2        2  0.85693  0.85693
Queue1   class1     5.2237  0.85693   6.0958   6.0958  0.85693  0.85693
Queue2   class1    0.73602  0.42846  0.85891  0.85891  0.85693  0.85693
Queue3   class1     5.2237  0.85693   6.0958   6.0958  0.85693  0.85693
Join     class1     5.6159        0   3.2768   3.2768   1.7139  0.85693
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.2 `fj_basic_closed`

The closed counterpart, on PS branches.
The example is built and solved as follows.

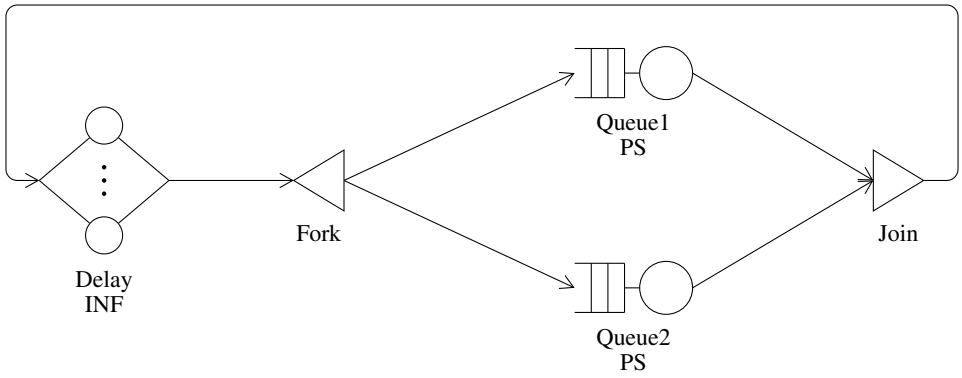

Figure 2.36: Topology of `fj_basic_closed`: a closed network over 3 queueing stations with a fork-join block.

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	1: class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – class1: Exp(1) Queue1 – class1: Exp(1) Queue2 – class1: Exp(1)
Solvers	JMT, MVA

Table 2.36: Features of `fj_basic_closed`.

```
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 5, delay)
delay.set_service(jobclass1, Exp(1.0))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, delay, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue1, join, 1.0)
P.set(jobclass1, jobclass1, queue2, join, 1.0)
P.set(jobclass1, jobclass1, join, delay, 1.0)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = fj_basic_closed()

solver = np.array([], dtype=object)
solver = np.append(solver, JMT(model, seed=23000))
solver = np.append(solver, MVA(model, method='amva', fork_join='ht'))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
```

```
Delay    class1    0.86625  0.86625  0.99611  0.99611  0.89442  0.89768
Queue1   class1     2.6475  0.88812   3.0945   3.0945  0.89768  0.89761
Queue2   class1     2.5863  0.89336    2.815    2.815  0.89768  0.89757
Join     class1     2.9633        0    1.685    1.685    1.724  0.88799

SOLVER: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    class1    0.94059  0.94059       1       1  0.94059  0.94059
Queue1   class1     2.7065  0.94051  2.8777  2.8777  0.94059  0.94051
Queue2   class1     2.7065  0.94051  2.8777  2.8777  0.94059  0.94051
Join     class1      2.706        0  1.4386  1.4386    1.881  0.94059
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.3 `fj_basic_nesting`

A fork inside a branch of another fork, with open and closed classes.

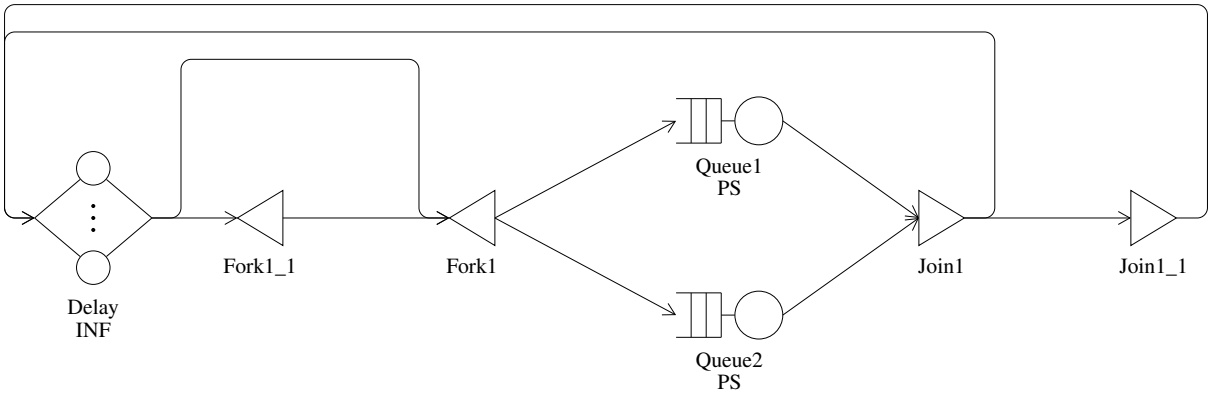

Figure 2.37: Topology of `fj_basic_nesting`: a closed network over 3 queueing stations with a fork-join block.

Nodes	7: Queue × 2, Delay, Fork × 2, Join × 2
Classes	2: class1 (closed, $N = 5$), class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – class1: Exp(0.25); class2: Exp(0.25) Queue1 – class1: Exp(1); class2: Exp(2) Queue2 – class1: Exp(0.75); class2: Exp(2)
Features	class switching on 2 links
Solvers	JMT, MVA

Table 2.37: Features of `fj_basic_nesting`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
model = Network('model')
# Nodes
delay = Delay(model, 'Delay')
fork1 = Fork(model, 'Fork1')
fork1.set_tasks_per_link(1)
fork11 = Fork(model, 'Fork1_1')
fork11.set_tasks_per_link(2)
join1 = Join(model, 'Join1', fork1)
join11 = Join(model, 'Join1_1', fork11)
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
# Classes
jobclass1 = ClosedClass(model, 'class1', 5, delay, 0)
jobclass2 = ClosedClass(model, 'class2', 2, delay, 0)
# Service times
delay.set_service(jobclass1, Exp(0.25))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(0.75))
delay.set_service(jobclass2, Exp(0.25))
queue1.set_service(jobclass2, Exp(2.0))
queue2.set_service(jobclass2, Exp(2.0))
# %%
# Routing

    ... (28 source lines omitted; full implementation: python/examples/basic/)

AvgTable = []
for s in solver:
    print(f'\nSOLVER: {s.get_name()}')
    avg_table = s.avg_table()
    print(avg_table)
    AvgTable.append(avg_table)
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    class1     2.0368   2.0368  4.0283  4.0283  0.50637  0.50487
Delay    class2    0.98642  0.98642  3.9138  3.9138  0.25427   0.2541
Join1    class1     2.2026        0  2.1777  2.1777  0.99731  0.50798
Join1    class2     1.0479        0  1.0011  1.0011   1.0445  0.51183
Join1_1  class2    0.54106        0  1.0555  1.0555  0.51183   0.2541
Queue1   class1     1.3447  0.49509  2.6485  2.6485  0.50487   0.5051
Queue1   class2    0.82969  0.26813  1.6297  1.6297  0.51178  0.51204
Queue2   class1     2.3299   0.6631  4.6827  4.6827  0.50487  0.50447
Queue2   class2     1.1149  0.25869  2.1947  2.1947  0.51178  0.51352

SOLVER: MVA

... (20 captured-output lines omitted; rerun the source example for the full output)

Delay    class2    0.93501  0.93501        4        4  0.23375  0.23375
Join1    class1     2.4047        0   2.7294   2.7294  0.88102  0.44051
Join1    class2    0.99834        0  0.85419  0.85419  0.93501   0.4675
Join1_1  class2    0.71667        0    1.533    1.533   0.4675  0.23375
Queue1   class1     1.2807  0.44051   2.9074   2.9074  0.44051  0.44051
Queue1   class2    0.68934  0.23375   1.4745   1.4745  0.23375   0.4675
Queue2   class1     2.9009  0.58734   6.5853   6.5853  0.44051  0.44051
Queue2   class2      1.179  0.23375   2.5219   2.5219  0.23375   0.4675
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.4 fj_basic_open

One fork, two parallel FCFS branches and a join, in an open network.

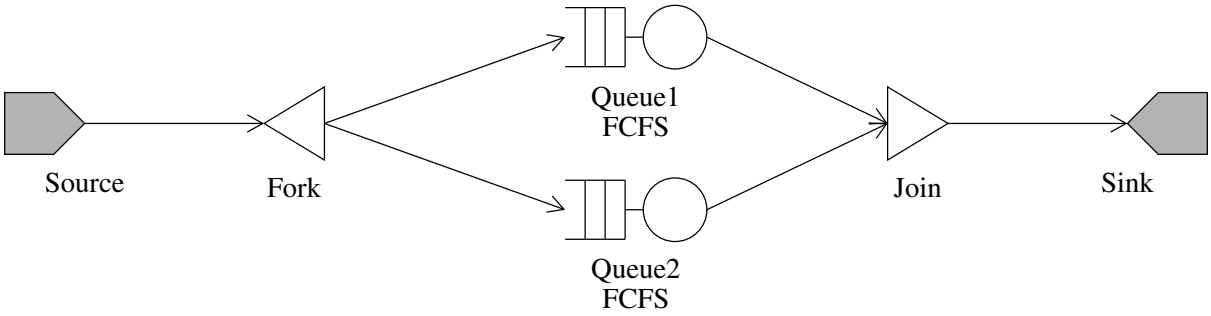

Figure 2.38: Topology of fj_basic_open: an open network over 2 queueing stations with a fork-join block.

Nodes	6: Source, Queue × 2, Fork, Join, Sink
Classes	1: class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – class1: Exp(0.05)
Service	Queue1 – class1: Exp(1) Queue2 – class1: Exp(2)
Solvers	JMT, MVA, LDES, MAM

Table 2.38: Features of fj_basic_open.

The example is built and solved as follows.

```
model = Network('model')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
sink = Sink(model, 'Sink')
jobclass1 = OpenClass(model, 'class1')
source.set_arrival(jobclass1, Exp(0.05))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(2.0))
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, source, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue1, join, 1.0)
P.set(jobclass1, jobclass1, queue2, join, 1.0)
P.set(jobclass1, jobclass1, join, sink, 1.0)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = fj_basic_open()

solver = np.array([], dtype=object)
solver = np.append(solver, JMT(model, seed=23000))
solver = np.append(solver, MVA(model))
solver = np.append(solver, LDES(model, seed=23000))
solver = np.append(solver, MAM(model, method='dec.source.mmap'))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass      QLen      Util    RespT   ResidT      ArvR      Tput
Source   class1          0         0        0        0         0  0.050494
Queue1   class1    0.051923  0.049524   1.0478   1.0478  0.050494  0.050812
Queue2   class1    0.026691  0.026329  0.50973  0.50973  0.050494  0.050896
Join     class1    0.044462         0  0.43237  0.43237   0.10395  0.050857

SOLVER: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (31 captured-output lines omitted; rerun the source example for the full output)

SOLVER: MAM
MAM FJ parametric decomposition completed in 3 iterations.
MAM analysis [method: dec.source.mmap; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s. Iterations: 3.
Station  JobClass      QLen   Util    RespT   ResidT  ArvR  Tput
Source   class1          0      0        0        0     0  0.05
Queue1   class1    0.052632   0.05   1.0526   1.0526  0.05  0.05
Queue2   class1    0.025641  0.025  0.51282  0.51282  0.05  0.05
Join     class1     0.04379      0   0.4379   0.4379   0.1  0.05
```

The rows repeat for each of the 4 solvers the script runs (JMT, MVA, LDES, MAM), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.5 `fj_complex_serial`

Several fork-join blocks in series inside a larger network.

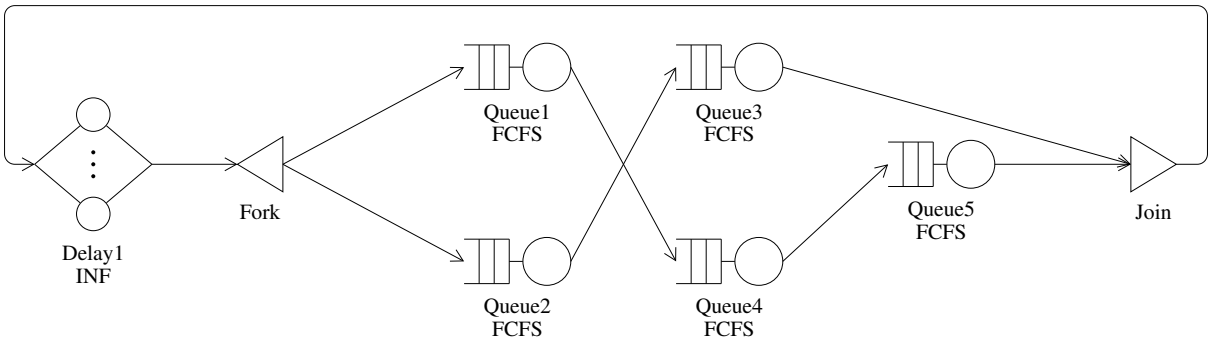

Figure 2.39: Topology of `fj_complex_serial`: a closed network over 6 queueing stations with a fork-join block.

Nodes	8: Queue × 5, Delay, Fork, Join
Classes	1: class1 (closed, $N = 10$)
Scheduling	Delay1: INF; Queue1: FCFS; Queue2: FCFS; Queue3: FCFS; Queue4: FCFS; Queue5: FCFS
Service	Delay1 – class1: Exp(0.5) Queue1 – class1: Exp(1) Queue2 – class1: Exp(2) Queue3 – class1: Exp(1) Queue4 – class1: Exp(3) Queue5 – class1: Exp(0.8)
Solvers	JMT, MVA

Table 2.39: Features of `fj_complex_serial`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
#Advanced Fork-Join Example 12#Complex fork-join with 2 parallel branches
model = Network('model')
# %%
# Create network
delay = Delay(model, 'Delay1')
queue1 = Queue(model,'Queue1',SchedStrategy.FCFS)
queue2 = Queue(model,'Queue2',SchedStrategy.FCFS)
queue3 = Queue(model,'Queue3',SchedStrategy.FCFS)
queue4 = Queue(model,'Queue4',SchedStrategy.FCFS)
queue5 = Queue(model,'Queue5',SchedStrategy.FCFS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
# %%
# Service configurations
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(2.0))
queue3.set_service(jobclass1, Exp(1.0))
queue4.set_service(jobclass1, Exp(3.0))
queue5.set_service(jobclass1, Exp(0.8))
delay.set_service(jobclass1, Exp(0.5))
# %%
# Set routing matrix with complex serial routing

    ... (13 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.5109   1.5109   1.9611   1.9611  0.76579  0.75823
Queue1   class1     2.5753  0.76218   3.3293   3.3293  0.75823  0.75929
Queue2   class1     0.5914  0.38738  0.79434  0.79434  0.75823  0.76578
Queue3   class1     2.7793  0.76354   3.4472   3.4472  0.76578  0.75992
Queue4   class1    0.32517  0.24092  0.41843  0.41843  0.75929  0.75933
Queue5   class1     5.3388  0.95413   7.2633   7.2633  0.75933  0.76579
Join     class1     5.3359        0   3.4698   3.4698   1.5125  0.75865
SOLVER 2: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (8 captured-output lines omitted; rerun the source example for the full output)

Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.3867   1.3867        2        2  0.69335  0.69335
Queue1   class1      2.142  0.69335   3.0894   3.0894  0.69335  0.69335
Queue2   class1    0.52374  0.34667  0.75538  0.75538  0.69335  0.69335
Queue3   class1      2.142  0.69335   3.0894   3.0894  0.69335  0.69335
Queue4   class1    0.29836  0.23112  0.43031  0.43031  0.69335  0.69335
Queue5   class1     5.6088  0.86669   8.0894   8.0894  0.69335  0.69335
Join     class1     6.7098        0   4.8387   4.8387   1.3867  0.69335
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.6 `fj_cs_multi_visits`

Class switching with several visits per branch.

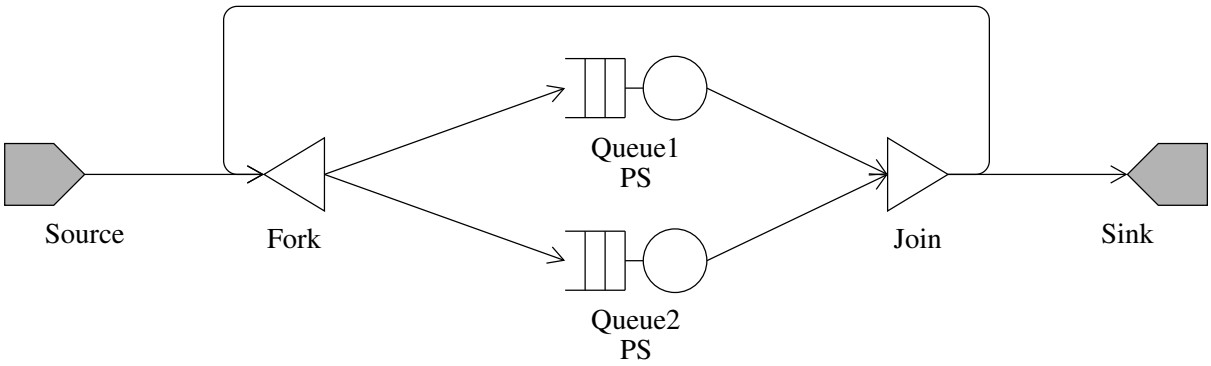

Figure 2.40: Topology of `fj_cs_multi_visits`: an open network over 2 queueing stations with a fork-join block.

Nodes	6: Source, Queue × 2, Fork, Join, Sink
Classes	2: class1 (open), class2 (open)
Scheduling	Queue1: PS; Queue2: PS
Arrivals	Source – class1: Exp(0.1); class2: Disabled
Service	Queue1 – class1: Exp(1); class2: Exp(1) Queue2 – class1: Exp(1); class2: Exp(1)
Features	class switching on 1 link
Solvers	MVA, JMT

Table 2.40: Features of `fj_cs_multi_visits`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join network example 8 - fork-join with multiple visits within the same chain
model = Network('model')
# %%
# Create network
source = Source(model,'Source')
queue1 = Queue(model,'Queue1',SchedStrategy.PS)
queue2 = Queue(model,'Queue2',SchedStrategy.PS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
sink = Sink(model,'Sink')
jobclass1 = OpenClass(model, 'class1')
jobclass2 = OpenClass(model, 'class2')
# %%
# Service configurations
source.set_arrival(jobclass1, Exp(0.1))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
queue1.set_service(jobclass2, Exp(1.0))
queue2.set_service(jobclass2, Exp(1.0))
# %%
# Set routing matrix with class switching - multiple visits
P = model.init_routing_matrix()
# Class 1 initial routing

    ... (18 source lines omitted; full implementation: python/examples/basic/)

avgTable = solver.avg_table()
print(avgTable)
```

```
solver = JMT(model, seed=23000)
print(f'SOLVER: {solver.get_name()}')
avgTable = solver.avg_table()
print(avgTable)
```

Running this edition prints

```
SOLVER: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (19 captured-output lines omitted; rerun the source example for the full output)

Station  JobClass     QLen     Util    RespT   ResidT      ArvR      Tput
Source   class1         0        0        0        0         0  0.099935
Queue1   class1   0.12819  0.09701   1.2308   1.2308  0.099935  0.099937
Queue1   class2    0.1269   0.1011    1.248    1.248  0.099931  0.099928
Queue2   class1   0.12642  0.10181    1.244    1.244  0.099935  0.099931
Queue2   class2    0.1243   0.1007   1.2318   1.2318  0.099931  0.099981
Join     class1   0.13493        0  0.66288  0.66288   0.19881  0.099931
Join     class2   0.12888        0   0.6401   0.6401   0.19882  0.099924
```

The rows repeat for each of the 2 solvers the script runs (MVA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.7 `fj_cs_postfork`

Class switching after the fork, on the sibling tasks.

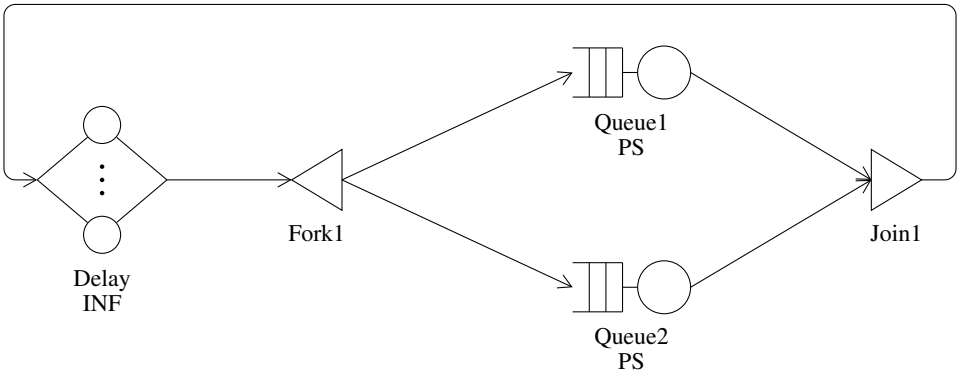

Figure 2.41: Topology of `fj_cs_postfork`: a closed network over 3 queueing stations with a fork-join block.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join network example 7
model = Network('model')
# %%
# Create network
```

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	2: class1 (closed, $N = 1$), class2 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – class1: Exp(0.25); class2: Exp(0.25) Queue1 – class1: Exp(2); class2: Exp(2) Queue2 – class1: Exp(2); class2: Exp(2)
Features	class switching on 3 links
Solvers	MVA, JMT

Table 2.41: Features of `fj_cs_postfork`.

```
delay = Delay(model,'Delay')
fork1 = Fork(model,'Fork1')
join1 = Join(model,'Join1',fork1)
queue1 = Queue(model,'Queue1',SchedStrategy.PS)
queue2 = Queue(model,'Queue2',SchedStrategy.PS)
jobclass1 = ClosedClass(model, 'class1', 1, delay, 0)
jobclass2 = ClosedClass(model, 'class2', 1, delay, 0)
# %%
# Service configurations
delay.set_service(jobclass1, Exp(0.25))
queue1.set_service(jobclass1, Exp(2.0))
queue2.set_service(jobclass1, Exp(2.0))
delay.set_service(jobclass2, Exp(0.25))
queue1.set_service(jobclass2, Exp(2.0))
queue2.set_service(jobclass2, Exp(2.0))
# %%
# Set routing matrix with class switching
P = model.init_routing_matrix()
# Class 1 routing

    ... (12 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [MVA(model), JMT(model, seed=23000)] # JMT has a bug on this one
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (14 captured-output lines omitted; rerun the source example for the full output)

JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    class1    0.83919  0.83919   4.0008   4.0008  0.20697  0.20977
Delay    class2    0.82894  0.82894   3.8875        0  0.20948  0.21192
Join1    class1    0.21374        0   0.2649   0.2649   0.8241  0.20774
Join1    class2          0        0        0        0      NaN  0.21192
Queue1   class1    0.21284  0.19531   0.5414   0.5414  0.41471  0.41463
Queue2   class1    0.22108  0.19926  0.54182  0.54182  0.41471   0.4147
```

The rows repeat for each of the 2 solvers the script runs (MVA, JMT), which is the point of the example:

the same model read by methods of different cost and different exactness.

2.6.8 `fj_cs_prefork`

Class switching before the fork.

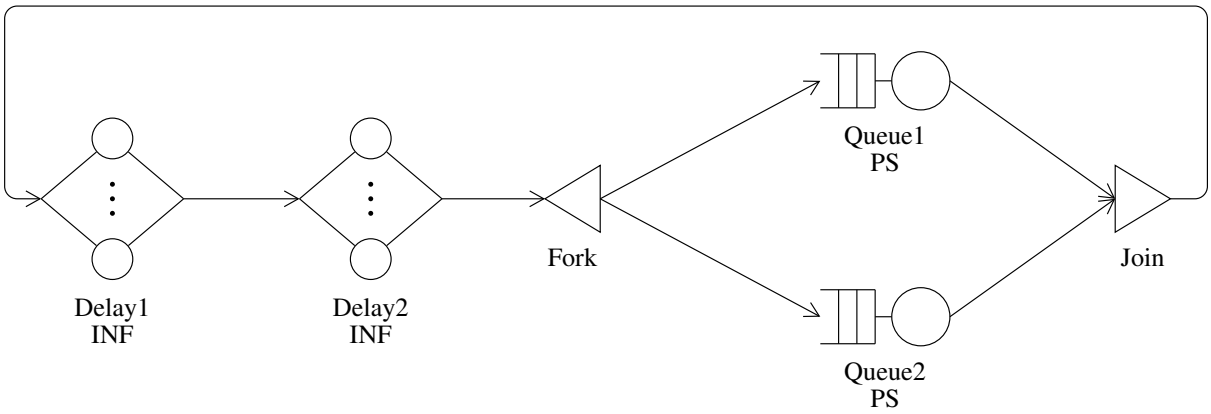

Figure 2.42: Topology of `fj_cs_prefork`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue × 2, Delay × 2, Fork, Join
Classes	2: class1 (closed, $N = 10$), class2 (closed, $N = 10$)
Scheduling	Delay1: INF; Delay2: INF; Queue1: PS; Queue2: PS
Service	Delay1 – class1: Exp(0.5); class2: Exp(0.5) Delay2 – class1: Disabled; class2: Exp(2) Queue1 – class1: Exp(1); class2: Disabled Queue2 – class1: Exp(1); class2: Disabled
Features	drop rule {'class1': 'waitingQueue'} at Delay2; drop rule {'class2': 'waitingQueue'} at Queue1; drop rule {'class2': 'waitingQueue'} at Queue2; class switching on 5 links
Solvers	JMT, MVA

Table 2.42: Features of `fj_cs_prefork`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join with Class Switching
model = Network('model')
# %%
# Create network
delay = Delay(model, 'Delay1')
delay2 = Delay(model, 'Delay2')
queue1 = Queue(model,'Queue1',SchedStrategy.PS)
queue2 = Queue(model,'Queue2',SchedStrategy.PS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
jobclass2 = ClosedClass(model, 'class2', 10, delay, 0)
# %%
# Service configurations
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
delay.set_service(jobclass1, Exp(0.5))
delay.set_service(jobclass2, Exp(0.5))
delay2.set_service(jobclass2, Exp(2.0))
# %%
# Set routing matrix with class switching
P = model.init_routing_matrix()
```

```
# Class switching routing: jobclass1 -> jobclass2 -> jobclass1

    ... (10 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.9083   1.9083   2.0226   2.0226   0.9777  0.97739
Delay2   class2    0.48212  0.48212  0.49535  0.49535  0.97739  0.97737
Queue1   class1     12.772  0.97594   12.462   12.462  0.97737  0.97823
Queue2   class1     8.7839  0.94243   8.9988   8.9988  0.97737  0.97343
Join     class1     13.632        0   7.0837   7.0837     1.92   0.9777
SOLVER 2: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (7 captured-output lines omitted; rerun the source example for the full output)

MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
Delay1   class1     1.8659   1.8659        2       2  0.93293  0.93293
Delay2   class2    0.46646  0.46646      0.5     0.5  0.93293  0.93293
Queue1   class1      11.87  0.93293   12.723  12.723  0.93293  0.93293
Queue2   class1      11.87  0.93293   12.723  12.723  0.93293  0.93293
Join     class1      11.87        0   6.3615  6.3615   1.8659  0.93293
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.9 `fj_deep_nesting`

Several nesting levels, the stress case for the join bookkeeping.

Nodes	9: Queue × 4, Delay, Fork × 2, Join × 2
Classes	1: class1 (closed, $N = 1$)
Scheduling	Delay1: INF; Queue1: FCFS; Queue2: FCFS; Queue3: FCFS; Queue4: FCFS
Service	Delay1 – class1: Exp(0.5) Queue1 – class1: Exp(1) Queue2 – class1: Exp(1) Queue3 – class1: Exp(2) Queue4 – class1: Exp(2)
Solvers	JMT, MVA

Table 2.43: Features of `fj_deep_nesting`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
```

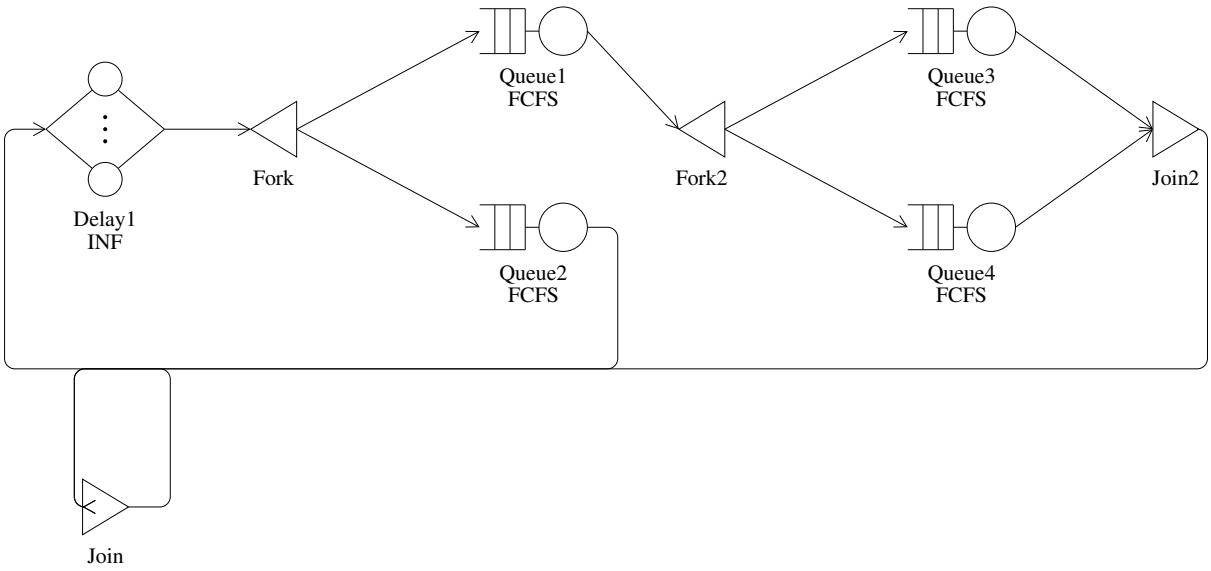

Figure 2.43: Topology of `fj_deep_nesting`: a closed network over 5 queueing stations with a fork-join block.

```
# Nested Fork-Join Network
model = Network('model')
# %%
# Create nested fork-join structure
delay = Delay(model, 'Delay1')
queue1 = Queue(model,'Queue1',SchedStrategy.FCFS)
queue2 = Queue(model,'Queue2',SchedStrategy.FCFS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
queue3 = Queue(model,'Queue3',SchedStrategy.FCFS)
queue4 = Queue(model,'Queue4',SchedStrategy.FCFS)
fork2 = Fork(model,'Fork2')
join2 = Join(model,'Join2', fork2)
jobclass1 = ClosedClass(model, 'class1', 1, delay, 0)
# %%
# Service configurations
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
delay.set_service(jobclass1, Exp(0.5))
queue3.set_service(jobclass1, Exp(2.0))
queue4.set_service(jobclass1, Exp(2.0))
# %%
# Nested fork-join routing

    ... (18 source lines omitted; full implementation: python/examples/basic/)

# Add MVA solver
solvers.append(MVA(model))
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
Delay1   class1    0.50301  0.50301   1.9945   1.9945   0.2517  0.24907
Queue1   class1     0.2479   0.2479   1.0192   1.0192  0.24907  0.24906
```

```
Queue2   class1    0.24341  0.24341  0.95781  0.95781  0.24907  0.25171
Join     class1    0.30974        0  0.62676  0.62676  0.50588  0.24906
Queue3   class1    0.12468  0.12468  0.50511  0.50511  0.24906  0.24844
Queue4   class1    0.12783  0.12783  0.50431  0.50431  0.24906  0.24907
Join2    class1    0.12602        0  0.24695  0.24695  0.50768  0.24845
SOLVER 2: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (20 captured-output lines omitted; rerun the source example for the full output)

Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
Delay1   class1    0.45192  0.45192        2         2  0.22596  0.22596
Queue1   class1    0.27306  0.22596   1.2084    1.2084  0.22596  0.22596
Queue2   class1    0.27306  0.22596   1.2084    1.2084  0.22596  0.22596
Join     class1    0.39228        0  0.86802   0.86802  0.45192  0.22596
Queue3   class1    0.12639  0.11298  0.55933   0.55933  0.22596  0.22596
Queue4   class1    0.12639  0.11298  0.55933   0.55933  0.22596  0.22596
Join2    class1    0.12639        0  0.22373   0.22373  0.45192  0.22596
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.10 `fj_delays`

Delay stations inside the branches.

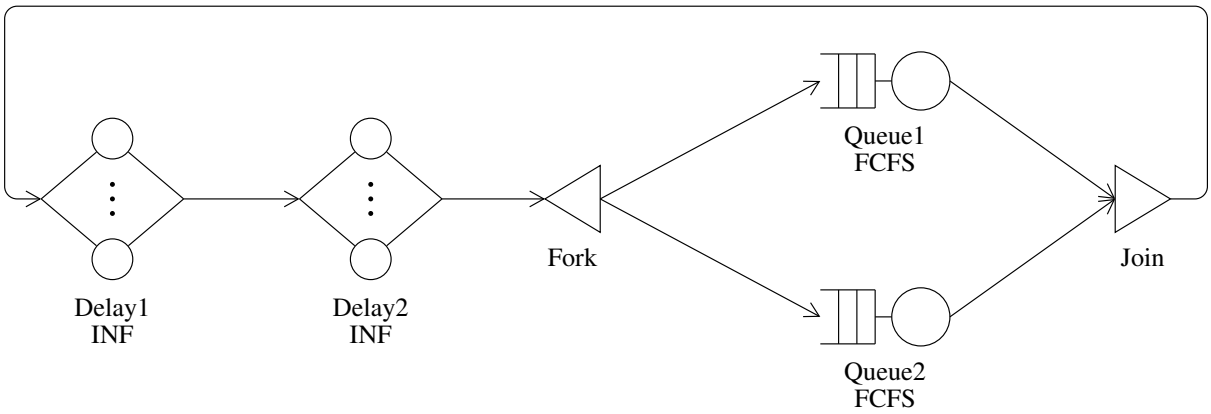

Figure 2.44: Topology of `fj_delays`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue × 2, Delay × 2, Fork, Join
Classes	1: class1 (closed, $N = 10$)
Scheduling	Delay1: INF; Delay2: INF; Queue1: FCFS; Queue2: FCFS
Service	Delay1 – class1: Exp(0.5) Delay2 – class1: Exp(2) Queue1 – class1: Exp(1) Queue2 – class1: Exp(1)
Solvers	JMT, MVA

Table 2.44: Features of `fj_delays`.

The example is built and solved as follows.

```
model = Network('model')
delay = Delay(model, 'Delay1')
delay2 = Delay(model, 'Delay2')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
```

```
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
delay.set_service(jobclass1, Exp(0.5))
delay2.set_service(jobclass1, Exp(2.0))
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, delay, delay2, 1.0)
P.set(jobclass1, jobclass1, delay2, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue1, join, 1.0)
P.set(jobclass1, jobclass1, queue2, join, 1.0)
P.set(jobclass1, jobclass1, join, delay, 1.0)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = fj_delays()

solver = np.array([], dtype=object)
solver = np.append(solver, JMT(model, seed=23000))
solver = np.append(solver, MVA(model))

avg_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_table[s] = solver[s].avg_table()
    print(avg_table[s])
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.9728   1.9728    2.035    2.035  0.96734  0.95415
Delay2   class1    0.48219  0.48219  0.49433  0.49433  0.95415  0.95412
Queue1   class1     6.0429  0.95704    6.081    6.081  0.95412  0.96494
Queue2   class1     5.1463  0.94515   5.7192   5.7192  0.95412  0.96616
Join     class1     3.9749        0   2.0321   2.0321   1.9436  0.96744

SOLVER: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (5 captured-output lines omitted; rerun the source example for the full output)

MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
Delay1   class1    1.7184   1.7184       2       2  0.85919  0.85919
Delay2   class1    0.4296   0.4296     0.5     0.5  0.85919  0.85919
Queue1   class1    5.3091  0.85919  6.1792  6.1792  0.85919  0.85919
Queue2   class1    5.3091  0.85919  6.1792  6.1792  0.85919  0.85919
Join     class1    5.3091        0  3.0896  3.0896   1.7184  0.85919
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.11 `fj_mixed_openclosed`

Open and closed classes through one fork-join block, with a class disabled on a branch.
The example is built and solved as follows.

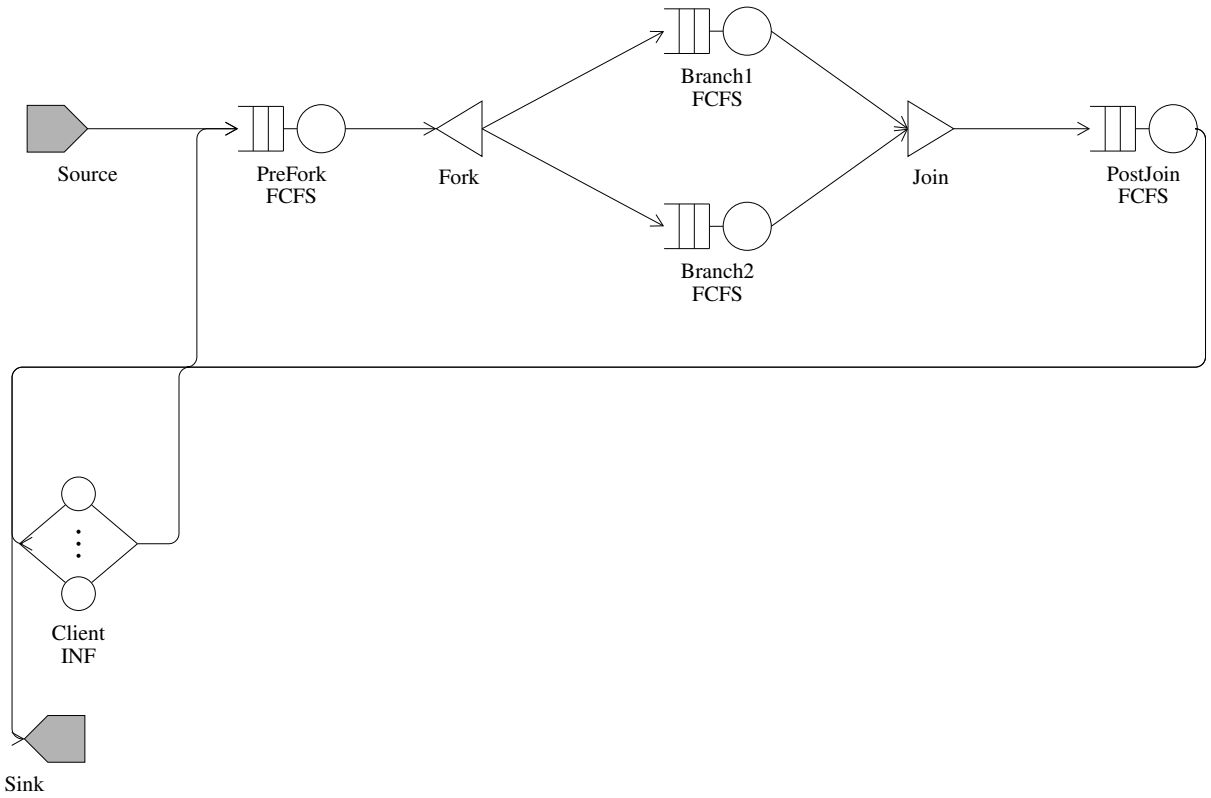

Figure 2.45: Topology of `fj_mixed_openclosed`: an open network over 5 queueing stations with a fork-join block.

Nodes	9: Source, Queue $\times$ 4, Delay, Fork, Join, Sink
Classes	2: Open (open), Closed (closed, $N = 1$)
Scheduling	Client: INF; PreFork: FCFS; Branch1: FCFS; Branch2: FCFS; PostJoin: FCFS
Arrivals	Source – Open: Exp(0.1); Closed: Disabled
Service	Client – Open: Disabled; Closed: Exp(1) PreFork – Open: Exp(5); Closed: Exp(5) Branch1 – Open: Exp(3.333); Closed: Exp(3.333) Branch2 – Open: Exp(2.5); Closed: Exp(2.5) PostJoin – Open: Exp(10); Closed: Exp(10)
Features	class switching on 2 links
Solvers	MVA, JMT

Table 2.45: Features of `fj_mixed_openclosed`.

```
model = Network('ForkJoinOpenClosed')
source = Source(model, 'Source')
client = Delay(model, 'Client')
prefork = Queue(model, 'PreFork', SchedStrategy.FCFS)
branch1 = Queue(model, 'Branch1', SchedStrategy.FCFS)
branch2 = Queue(model, 'Branch2', SchedStrategy.FCFS)
postjoin = Queue(model, 'PostJoin', SchedStrategy.FCFS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Open')
cclass = ClosedClass(model, 'Closed', 1, client)
source.set_arrival(oclass, Exp(0.1))
client.set_service(cclass, Exp.fit_mean(1.0))
client.set_service(oclass, Disabled())
for station, mean in ((prefork, 0.2), (branch1, 0.3), (branch2, 0.4), (postjoin, 0.1)):
    station.set_service(oclass, Exp.fit_mean(mean))
    station.set_service(cclass, Exp.fit_mean(mean))
P = model.init_routing_matrix()
for jobclass, entry, exit_node in ((oclass, source, sink), (cclass, client, client)):
    P.set(jobclass, jobclass, entry, prefork, 1.0)
    P.set(jobclass, jobclass, prefork, fork, 1.0)
    P.set(jobclass, jobclass, fork, branch1, 1.0)
    P.set(jobclass, jobclass, fork, branch2, 1.0)
    P.set(jobclass, jobclass, branch1, join, 1.0)
    P.set(jobclass, jobclass, branch2, join, 1.0)

    ... (5 source lines omitted; full implementation: python/examples/basic/)

model = fj_mixed_openclosed()

print('\nMVA Results:')
print(SolverMVA(model).get_avg_table())
print('\nJMT Results:')
print(SolverJMT(model, seed=23000, samples=200000).get_avg_table())
```

Running this edition prints

```
MVA Results:
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (30 captured-output lines omitted; rerun the source example for the full output)

Branch1   Open      0.035936   0.029862  0.36021  0.36021  0.10008  0.10008
Branch1   Closed     0.16655    0.16185  0.30807  0.30807  0.54074  0.54099
Branch2   Open       0.05106   0.039888  0.50925  0.50925  0.10008  0.10008
Branch2   Closed     0.22565    0.21675  0.41709  0.41709  0.54074  0.54099
PostJoin  Open      0.010513  0.0099568  0.10573  0.10573  0.10008  0.10008
PostJoin  Closed    0.054457    0.05398  0.10078  0.10078  0.54099  0.54099
Join      Open      0.042761          0  0.21454  0.21454  0.19947  0.10008
Join      Closed     0.19875          0  0.18427  0.18427   1.0827  0.54099
```

The rows repeat for each of the 2 solvers the script runs (MVA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.12 fj_nojoin

A fork with no join, so the siblings leave through the sink independently.

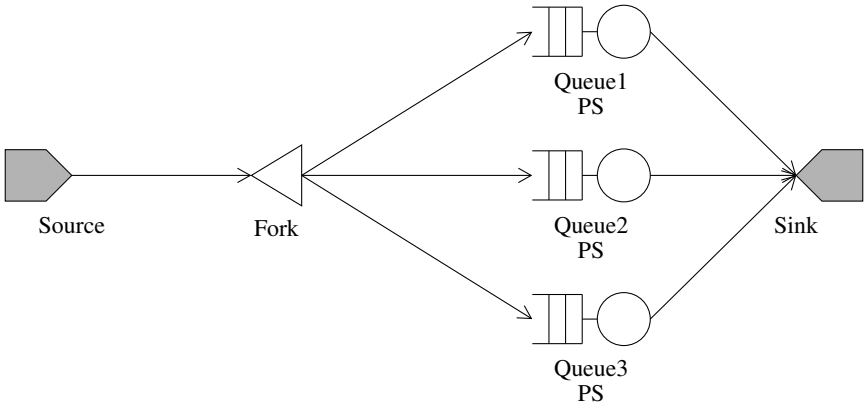

Figure 2.46: Topology of fj_nojoin: an open network over 3 queueing stations with a fork-join block.

Nodes	6: Source, Queue × 3, Fork, Sink
Classes	1: class1 (open)
Scheduling	Queue1: PS; Queue2: PS; Queue3: PS
Arrivals	Source – class1: Exp(0.5)
Service	Queue1 – class1: Exp(1) Queue2 – class1: Exp(2) Queue3 – class1: Exp(3)
Solvers	MVA, JMT

Table 2.46: Features of fj_nojoin.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join network example 4
model = Network('model')
# %%
# Create network
source = Source(model,'Source')
queue1 = Queue(model,'Queue1',SchedStrategy.PS)
queue2 = Queue(model,'Queue2',SchedStrategy.PS)
queue3 = Queue(model,'Queue3',SchedStrategy.PS)
fork = Fork(model,'Fork')
sink = Sink(model,'Sink')
jobclass1 = OpenClass(model, 'class1')
# %%
# Service configurations
source.set_arrival(jobclass1, Exp(0.5))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(2.0))
queue3.set_service(jobclass1, Exp(3.0))
# %%
# Pure fork routing (no join)
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, source, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)

    ... (6 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [MVA(model), JMT(model, seed=23000)]
```

```
for i, solver in enumerate(solvers):
    print(f'\nSOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (15 captured-output lines omitted; rerun the source example for the full output)

JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR     Tput
Source   class1         0        0        0        0       0   0.5062
Queue1   class1   0.97824  0.49952   1.9229   1.9229  0.5062  0.50103
Queue2   class1   0.32722  0.24452   0.6673   0.6673  0.5062  0.50398
Queue3   class1    0.2038  0.16807  0.41146  0.41146  0.5062  0.50621
```

The rows repeat for each of the 2 solvers the script runs (MVA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.13 `fj_route_overlap`

Branches that share a station, so their visits overlap.

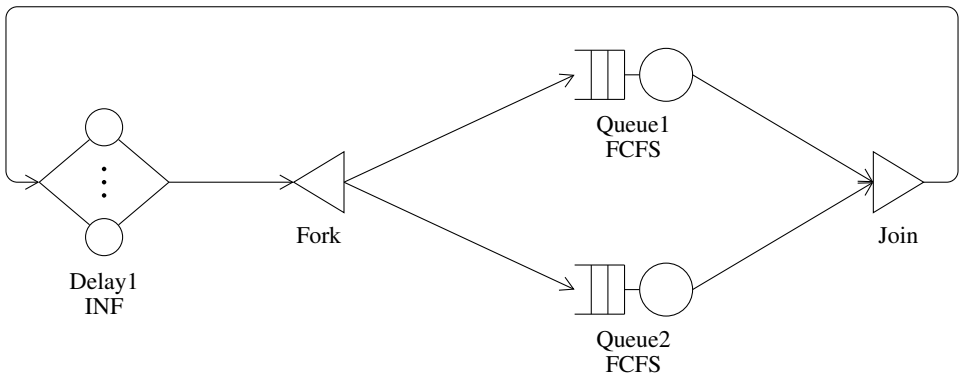

Figure 2.47: Topology of `fj_route_overlap`: a closed network over 3 queueing stations with a fork-join block.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join network example 9
model = Network('model')
# %%
```

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	2: class1 (closed, $N = 10$), class2 (closed, $N = 10$)
Scheduling	Delay1: INF; Queue1: FCFS; Queue2: FCFS
Service	Delay1 – class1: Exp(0.5); class2: Exp(0.2) Queue1 – class1: Exp(1); class2: Exp(1) Queue2 – class1: Exp(2); class2: Exp(2)
Solvers	JMT, MVA

Table 2.47: Features of `fj_route_overlap`.

```
# Create network
delay = Delay(model, 'Delay1')
queue1 = Queue(model,'Queue1',SchedStrategy.FCFS)
queue2 = Queue(model,'Queue2',SchedStrategy.FCFS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
jobclass2 = ClosedClass(model, 'class2', 10, delay, 0)
# %%
# Service configurations
# Class 1
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(2.0))
delay.set_service(jobclass1, Exp(0.5))
# Class 2
queue1.set_service(jobclass2, Exp(1.0))
queue2.set_service(jobclass2, Exp(2.0))
delay.set_service(jobclass2, Exp(0.2))
# %%
# Set routing matrix

    ... (17 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   class1     1.0434   1.0434   1.9568   1.9568  0.53561  0.53361
Delay1   class2     2.2482   2.2482   4.9237   4.9237  0.46316  0.46313
Queue1   class1     8.9566  0.53833   16.567   16.567  0.53361  0.53561
Queue1   class2     7.7518  0.45808   16.743   16.743  0.46313   0.4626
Queue2   class1    0.52817  0.27261  0.99797  0.99797  0.53361   0.5331
Queue2   class2    0.47122  0.22608   1.0143   1.0143  0.46313   0.4629
Join     class1     8.4186        0   7.8315   7.8315   1.0916  0.53561
Join     class2     7.2339        0   7.7245   7.7245  0.91758   0.4626
SOLVER 2: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (34 captured-output lines omitted; rerun the source example for the full output)

Delay1   class1     1.0241   1.0241        2        2  0.51206  0.51206
Delay1   class2      2.215    2.215        5        5    0.443    0.443
Queue1   class1     9.0096  0.51206   17.595   17.595  0.51206  0.51206
Queue1   class2     7.8075    0.443   17.624   17.624    0.443    0.443
Queue2   class1    0.48432  0.25603  0.94582  0.94582  0.51206  0.51206
Queue2   class2    0.41936   0.2215  0.94663  0.94663    0.443    0.443
Join     class1     8.5747        0   8.3727   8.3727   1.0241  0.51206
Join     class2     7.4309        0    8.387    8.387    0.886    0.443
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.14 `fj_serialfjs_closed`

The closed version.

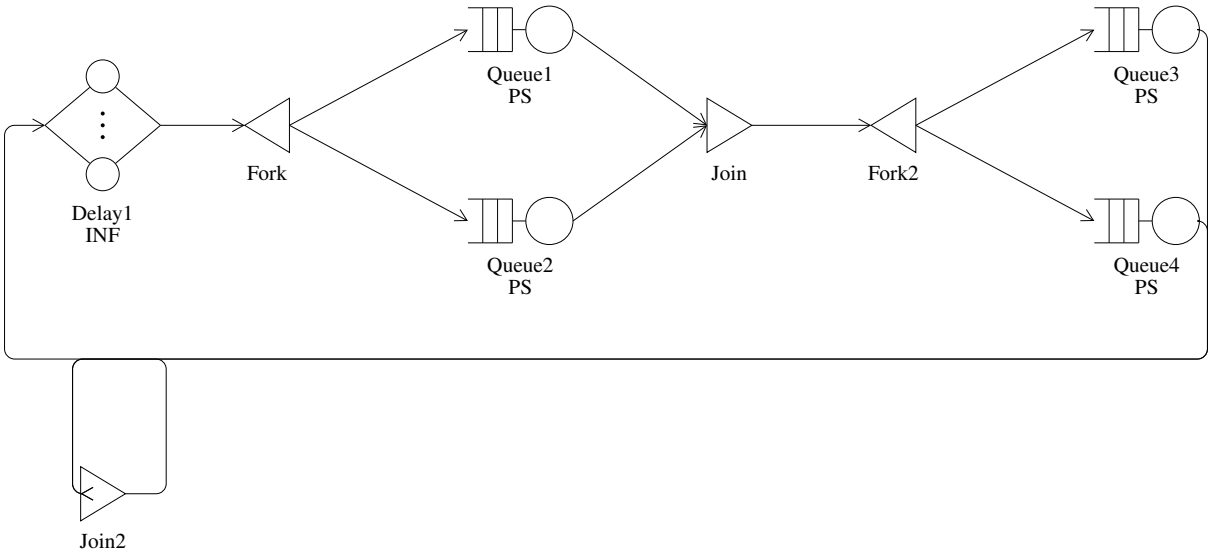

Figure 2.48: Topology of `fj_serialfjs_closed`: a closed network over 5 queueing stations with a fork-join block.

Nodes	9: Queue × 4, Delay, Fork × 2, Join × 2
Classes	1: class1 (closed, $N = 10$)
Scheduling	Delay1: INF; Queue1: PS; Queue2: PS; Queue3: PS; Queue4: PS
Service	Delay1 – class1: Exp(0.5) Queue1 – class1: Exp(1) Queue2 – class1: Exp(1) Queue3 – class1: Exp(1) Queue4 – class1: Exp(1)
Solvers	JMT, MVA

Table 2.48: Features of `fj_serialfjs_closed`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Series of Fork-Join Networks
model = Network('model')
# %%
# Create series of fork-join stages
delay = Delay(model, 'Delay1')
queue1 = Queue(model,'Queue1',SchedStrategy.PS)
queue2 = Queue(model,'Queue2',SchedStrategy.PS)
fork = Fork(model,'Fork')
join = Join(model,'Join', fork)
queue3 = Queue(model,'Queue3',SchedStrategy.PS)
queue4 = Queue(model,'Queue4',SchedStrategy.PS)
fork2 = Fork(model,'Fork2')
join2 = Join(model,'Join2', fork2)
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
# %%
# Service configurations
queue1.set_service(jobclass1, Exp(1.0))
```

```
queue2.set_service(jobclass1, Exp(1.0))
delay.set_service(jobclass1, Exp(0.5))
queue3.set_service(jobclass1, Exp(1.0))
queue4.set_service(jobclass1, Exp(1.0))
# %%
# Series fork-join routing

    ... (17 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Delay1   class1    1.5478   1.5478  2.0184  2.0184  0.80819  0.81366
Queue1   class1    2.6984  0.81744  3.4341  3.4341  0.81366  0.80809
Queue2   class1    2.8035  0.80343  3.4227  3.4227  0.81366  0.81267
Join     class1    3.1856        0  1.9318  1.9318    1.588  0.80888
Queue3   class1    2.5598  0.80584  3.1944  3.1944  0.80888  0.80815
Queue4   class1    2.6804  0.80079  3.3994  3.3994  0.80888  0.80952
Join2    class1    2.8828        0  1.7538  1.7538   1.6209    0.808
SOLVER 2: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
     >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
     >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
     >s.

... (7 captured-output lines omitted; rerun the source example for the full output)

Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Delay1   class1    1.5101   1.5101        2       2  0.75503  0.75503
Queue1   class1    2.8654  0.75503  3.7951  3.7951  0.75503  0.75503
Queue2   class1    2.8654  0.75503  3.7951  3.7951  0.75503  0.75503
Join     class1    2.8654        0  1.8975  1.8975   1.5101  0.75503
Queue3   class1    2.8654  0.75503  3.7951  3.7951  0.75503  0.75503
Queue4   class1    2.8654  0.75503  3.7951  3.7951  0.75503  0.75503
Join2    class1    2.8654        0  1.8975  1.8975   1.5101  0.75503
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.15 `fj_serialfjs_open`

A chain of fork-join blocks, open.

Nodes	10: Source, Queue × 4, Fork × 2, Join × 2, Sink
Classes	1: class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS; Queue4: FCFS
Arrivals	Source – class1: Exp(0.4)
Service	Queue1 – class1: Exp(1) Queue2 – class1: Exp(1) Queue3 – class1: Exp(1) Queue4 – class1: Exp(1)
Solvers	JMT, MVA

Table 2.49: Features of `fj_serialfjs_open`.

The example is built and solved as follows.

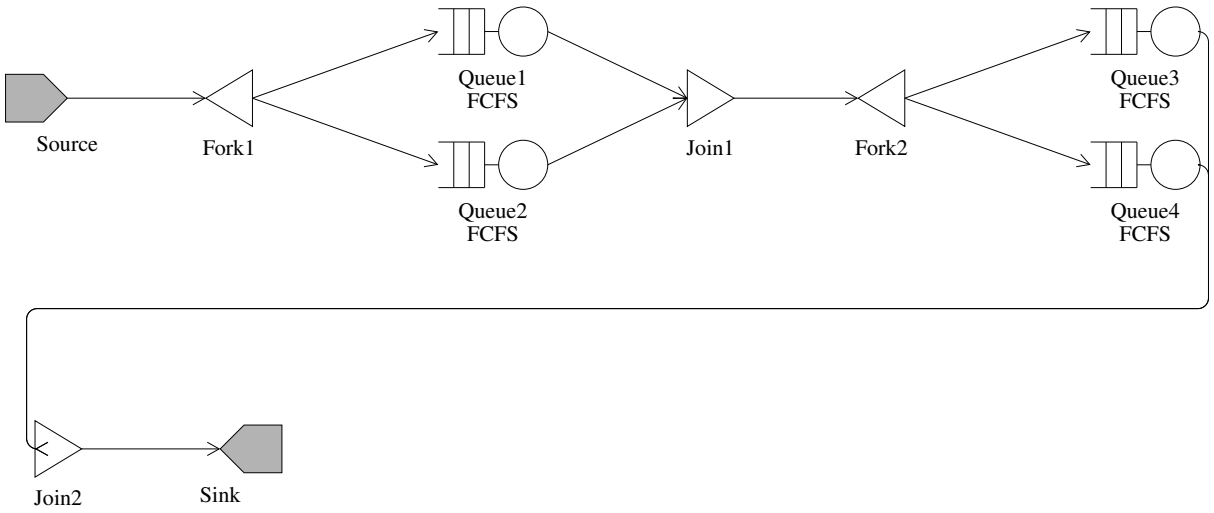

Figure 2.49: Topology of `fj_serialfjs_open`: an open network over 4 queueing stations with a fork-join block.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join network example 6
model = Network('model')
# %%
# Create network
source = Source(model,'Source')
queue1 = Queue(model,'Queue1',SchedStrategy.FCFS)
queue2 = Queue(model,'Queue2',SchedStrategy.FCFS)
fork1 = Fork(model,'Fork1')
join1 = Join(model,'Join1',fork1)
queue3 = Queue(model,'Queue3',SchedStrategy.FCFS)
queue4 = Queue(model,'Queue4',SchedStrategy.FCFS)
fork2 = Fork(model,'Fork2')
join2 = Join(model,'Join2',fork2)
sink = Sink(model,'Sink')
jobclass1 = OpenClass(model, 'class1')
# %%
# Service configurations
source.set_arrival(jobclass1, Exp(0.4))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(1.0))
queue3.set_service(jobclass1, Exp(1.0))
queue4.set_service(jobclass1, Exp(1.0))
# %%

    ... (15 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
Source   class1          0        0        0        0        0  0.40474
```

```
Queue1   class1    0.68628  0.40101   1.6675   1.6675  0.40474   0.4092
Queue2   class1    0.69566  0.40012   1.7616   1.7616  0.40474  0.40471
Join1    class1    0.59155        0  0.76312  0.76312  0.83292  0.40867
Queue3   class1    0.68976  0.41292   1.6883   1.6883  0.40867  0.40872
Queue4   class1     0.6731  0.41363   1.6489   1.6489  0.40867  0.41236
Join2    class1    0.61305        0   0.7252   0.7252  0.83343  0.41225
SOLVER 2: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (20 captured-output lines omitted; rerun the source example for the full output)

Station  JobClass     QLen  Util    RespT   ResidT  ArvR  Tput
Source   class1          0     0        0        0     0   0.4
Queue1   class1    0.66666   0.4   1.6667   1.6667   0.4   0.4
Queue2   class1    0.66666   0.4   1.6667   1.6667   0.4   0.4
Join1    class1    0.66666     0  0.83333  0.83333   0.8   0.4
Queue3   class1    0.66666   0.4   1.6667   1.6667   0.4   0.4
Queue4   class1    0.66666   0.4   1.6667   1.6667   0.4   0.4
Join2    class1    0.66666     0  0.83333  0.83333   0.8   0.4
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.16 `fj_threebranches`

Three parallel branches rather than two.

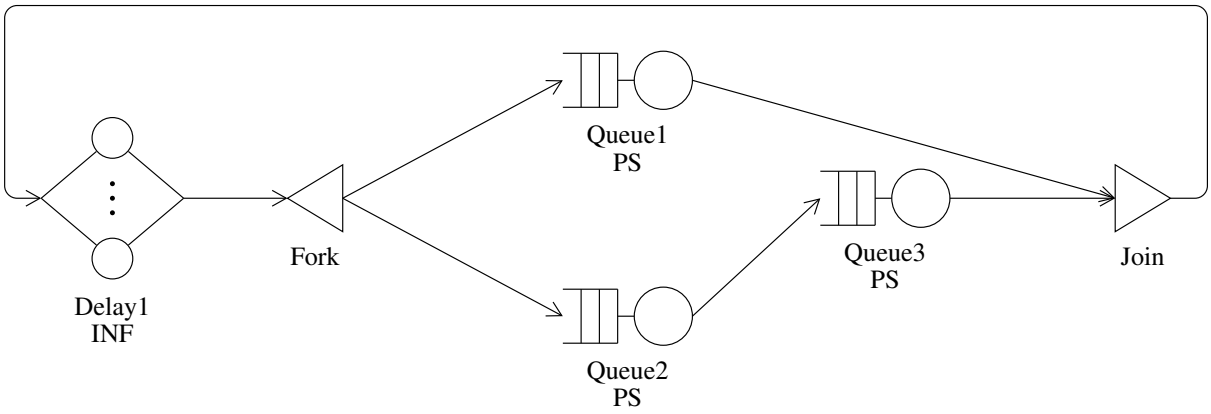

Figure 2.50: Topology of `fj_threebranches`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue × 3, Delay, Fork, Join
Classes	2: class1 (closed, $N = 10$), class2 (closed, $N = 10$)
Scheduling	Delay1: INF; Queue1: PS; Queue2: PS; Queue3: PS
Service	Delay1 – class1: Exp(0.5); class2: Exp(0.8) Queue1 – class1: Exp(1.5); class2: Exp(2.8) Queue2 – class1: Exp(1.1); class2: Exp(3) Queue3 – class1: Exp(2.5); class2: Exp(1)
Solvers	MVA

Table 2.50: Features of `fj_threebranches`.

The example is built and solved as follows.

```
model = Network('model')
delay = Delay(model, 'Delay1')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue3 = Queue(model, 'Queue3', SchedStrategy.PS)
fork = Fork(model, 'Fork')
```

```
join = Join(model, 'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 10, delay, 0)
jobclass2 = ClosedClass(model, 'class2', 10, delay, 0)
# Class 1 service times
queue1.set_service(jobclass1, Exp(1.5))
queue2.set_service(jobclass1, Exp(1.1))
queue3.set_service(jobclass1, Exp(2.5))
delay.set_service(jobclass1, Exp(0.5))
# Class 2 service times
queue1.set_service(jobclass2, Exp(2.8))
queue2.set_service(jobclass2, Exp(3.0))
queue3.set_service(jobclass2, Exp(1.0))
delay.set_service(jobclass2, Exp(0.8))
P = model.init_routing_matrix()
# Class 1 routing
P.set(jobclass1, jobclass1, delay, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue2, queue3, 1.0)
P.set(jobclass1, jobclass1, queue3, join, 1.0)

    ... (13 source lines omitted; full implementation: python/examples/basic/)

model = fj_threebranches()

solver = MVA(model)
print(f'SOLVER: {solver.get_name()}')
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
SOLVER: MVA
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.
MVA analysis [method: egflin; type: approximate, deterministic; lang: python; env: python] completed in <t
    >s.

... (9 captured-output lines omitted; rerun the source example for the full output)

Queue1   class1     1.6413  0.48637  2.2497  2.2497  0.72955  0.72955
Queue1   class2    0.77382  0.22787  1.2128  1.2128  0.63804  0.63804
Queue2   class1     4.8036  0.66323  6.5844  6.5844  0.72955  0.72955
Queue2   class2     1.5573  0.21268  2.4407  2.4407  0.63804  0.63804
Queue3   class1      3.529  0.29182  4.8372  4.8372  0.72955  0.72955
Queue3   class2     7.6515  0.63804  11.992  11.992  0.63804  0.63804
Join     class1     7.2315        0  4.9561  4.9561   1.4591  0.72955
Join     class2      8.555        0  6.7041  6.7041   1.2761  0.63804
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.6.17 `fj_tiny_closed`

The smallest closed fork-join model, small enough for an exact CTMC solve.

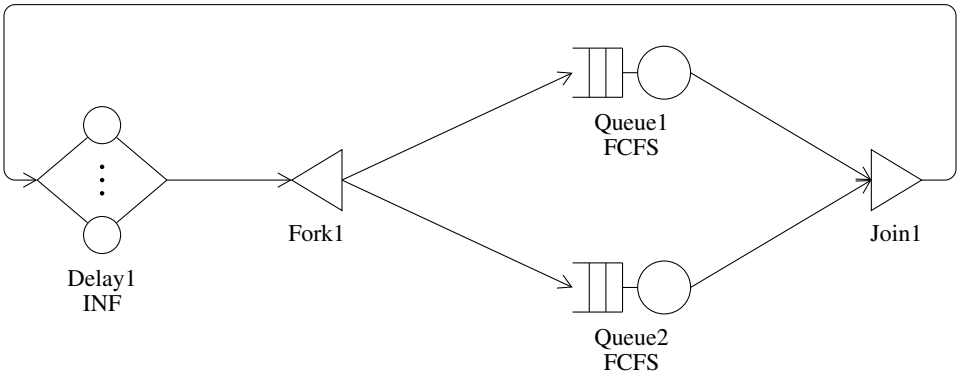

Figure 2.51: Topology of `fj_tiny_closed`: a closed network over 3 queueing stations with a fork-join block.

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	1: Class1 (closed, $N = 1$)
Scheduling	Delay1: INF; Queue1: FCFS; Queue2: FCFS
Service	Delay1 – Class1: Exp(1) Queue1 – Class1: Exp(2) Queue2 – Class1: Exp(3)
Solvers	CTMC, SSA, JMT

Table 2.51: Features of `fj_tiny_closed`.

The example is built and solved as follows.

```
model = Network('model')
delay = Delay(model, 'Delay1')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
fork = Fork(model, 'Fork1')
join = Join(model, 'Join1', fork)
jobclass1 = ClosedClass(model, 'Class1', 1, delay)
delay.set_service(jobclass1, Exp(1.0))
queue1.set_service(jobclass1, Exp(2.0))
queue2.set_service(jobclass1, Exp(3.0))
P = model.init_routing_matrix()
P.set(jobclass1, jobclass1, delay, fork, 1.0)
P.set(jobclass1, jobclass1, fork, queue1, 1.0)
P.set(jobclass1, jobclass1, fork, queue2, 1.0)
P.set(jobclass1, jobclass1, queue1, join, 1.0)
P.set(jobclass1, jobclass1, queue2, join, 1.0)
P.set(jobclass1, jobclass1, join, delay, 1.0)
model.link(P)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = fj_tiny_closed()

solver = [CTMC(model),
          SSA(model, seed=23000, samples=50000),
          JMT(model, seed=23000)]

for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name().replace("Solver", "")}')
    print(solver[s].avg_table())
```

Running this edition prints

```
SOLVER: CTMC
  Stochcomp: 2 immediate states eliminated, 4 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   Class1    0.61224  0.61224        1        1  0.61224  0.61224
Queue1   Class1    0.30612  0.30612      0.5      0.5  0.61224  0.61224
Queue2   Class1    0.20408  0.20408  0.33333  0.33333  0.61224  0.61224
Join1    Class1    0.26531        0  0.21667  0.21667   1.2245  0.61224

SOLVER: SSA
SSA analysis [method: default/serial; type: approximate, randomized; lang: python; env: python] completed
    in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   Class1    0.60986  0.60986  0.98094  0.98094  0.62171  0.60986
Queue1   Class1    0.30719  0.30759  0.50371  0.50371  0.60986  0.61439
Queue2   Class1    0.20832  0.20506  0.34159  0.34159  0.60986  0.62497

... (3 captured-output lines omitted; rerun the source example for the full output)

JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay1   Class1    0.61067  0.61067  0.98692  0.98692  0.61699  0.61569
Queue1   Class1    0.31288  0.31288  0.49922  0.49922  0.61569  0.61578
Queue2   Class1     0.2034   0.2034  0.33361  0.33361  0.61569  0.61713
Join1    Class1    0.26125        0  0.21207  0.21207   1.2554  0.61576
```

The rows repeat for each of the 3 solvers the script runs (CTMC, SSA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.18 `fj_twoclasses_forked`

Two classes forked independently through the same block.

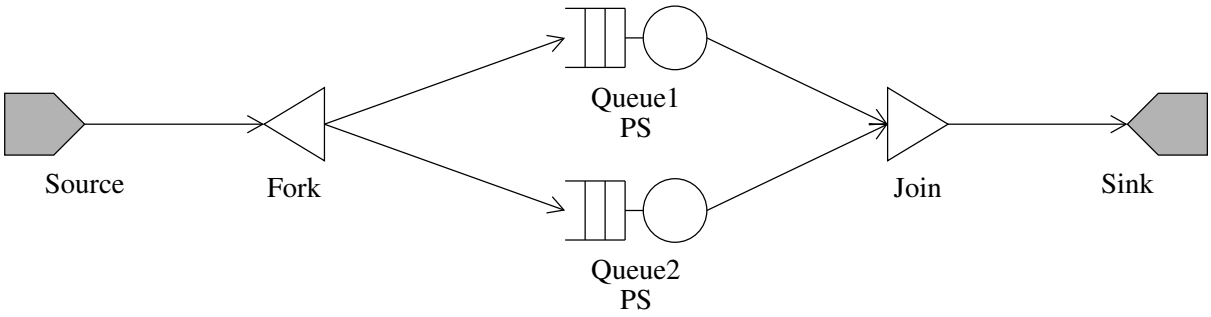

Figure 2.52: Topology of `fj_twoclasses_forked`: an open network over 2 queueing stations with a fork-join block.

Nodes	6: Source, Queue × 2, Fork, Join, Sink
Classes	2: class1 (open), class2 (open)
Scheduling	Queue1: PS; Queue2: PS
Arrivals	Source – class1: Exp(0.25); class2: Exp(0.25)
Service	Queue1 – class1: Exp(1); class2: Immediate Queue2 – class1: Exp(0.75); class2: Exp(2)
Solvers	JMT, MVA

Table 2.52: Features of `fj_twoclasses_forked`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Fork-Join queueing network
model = Network('model')
# %%
# Create network
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
fork = Fork(model, 'Fork')
fork.set_tasks_per_link(2)
join = Join(model, 'Join', fork)
sink = Sink(model, 'Sink')
# %%
# Create job classes
jobclass1 = OpenClass(model, 'class1')
jobclass2 = OpenClass(model, 'class2')
# %%
#Set arrival and service processes
# Class 1
source.set_arrival(jobclass1, Exp(0.25))
queue1.set_service(jobclass1, Exp(1.0))
queue2.set_service(jobclass1, Exp(0.75))
# Class 2
source.set_arrival(jobclass2, Exp(0.25))

    ... (19 source lines omitted; full implementation: python/examples/basic/)

# Solve with multiple methods
solvers = [ JMT(model, seed=23000), MVA(model)]
for i, solver in enumerate(solvers):
    print(f'SOLVER {i+1}: {solver.get_name()}')
    avgTable = solver.avg_table()
    print(avgTable)
```

Running this edition prints

```
SOLVER 1: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Source   class1         0        0        0       0        0  0.25128
Source   class2         0        0        0       0        0  0.25184
Queue1   class1    1.4252  0.49231   2.9402  2.9402  0.50253  0.49004
Queue1   class2         0        0        0       0  0.50531  0.50531
Queue2   class1    8.7958  0.65567  17.406  17.406  0.50253  0.50863
Queue2   class2    3.4512  0.25836   6.7395  6.7395  0.50531  0.50463
Join     class1    14.468        0  14.673  14.673   0.9868  0.24763
Join     class2     6.754        0   6.305   6.305    1.028  0.25154
SOLVER 2: MVA
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.

... (26 captured-output lines omitted; rerun the source example for the full output)

Source   class1         0        0       0       0     0  0.25
Source   class2         0        0       0       0     0  0.25
Queue1   class1         1      0.5       2       2  0.25   0.5
Queue1   class2         0        0       0       0  0.25   0.5
Queue2   class1         8  0.66667      16      16  0.25   0.5
Queue2   class2         3     0.25       6       6  0.25   0.5
Join     class1    23.444        0  23.444  23.444     1  0.25
Join     class2         9        0       9       9     1  0.25
```

The rows repeat for each of the 2 solvers the script runs (JMT, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.6.19 `fj_variable_fanout`

A forking level that is not one fixed number: a per-link vector, a random degree, or a branch that fires only part of the time.

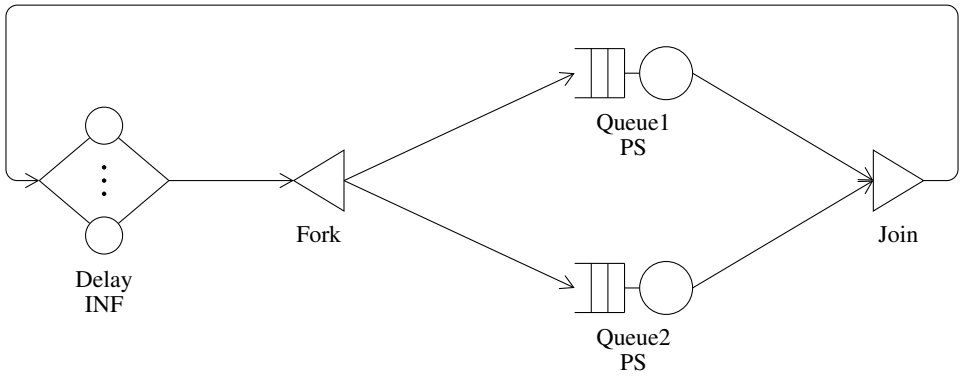

Figure 2.53: Topology of `fj_variable_fanout`: a closed network over 3 queueing stations with a fork-join block.

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	1: class1 (closed, $N = 4$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – class1: Exp(1) Queue1 – class1: Exp(2) Queue2 – class1: Exp(2)
Solvers	JMT

Table 2.53: Features of `fj_variable_fanout`.

The example is built and solved as follows.

```
'fixed'   the classic fork, one task on each of two links (the baseline)
'vector'  three tasks towards Queue2 and one towards Queue1
'random'  one or three tasks per link, each with probability 1/2
'prob'    the branch towards Queue2 fires only half the time
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
jobclass1 = ClosedClass(model, 'class1', 4, delay)
delay.set_service(jobclass1, Exp(1.0))
queue1.set_service(jobclass1, Exp(2.0))
queue2.set_service(jobclass1, Exp(2.0))
if mode == 'fixed':
    pass  # the baseline every other mode is compared against
elif mode == 'vector':
    fork.setTasksPerLink(3, jobclass1, queue2)
elif mode == 'random':
    fork.setTasksPerLinkDistribution(jobclass1, DiscreteSampler([0.5, 0.5], [1, 3]))
elif mode == 'prob':
    # an uncertain branch needs a Join that does not wait for it
    join.setStrategy(jobclass1, JoinStrategy.PARTIAL)
    join.setRequired(jobclass1, 1)
    fork.setBranchProbability(jobclass1, queue2, 0.5)
else:

    ... (9 source lines omitted; full implementation: python/examples/basic/)

return model
GlobalConstants.set_verbose(VerboseLevel.STD)
for mode in ('fixed', 'vector', 'random', 'prob'):
```

```
    model = fj_variable_fanout(mode)
    print('\nmode = %s' % mode)
    print(JMT(model, seed=23000).avg_table())
```

Running this edition prints

```
mode = fixed
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Delay    class1    1.5341   1.5341   1.0133   1.0133  1.5446  1.5448
Queue1   class1    1.5889   0.7622   1.0174   1.0174  1.5448  1.5455
Queue2   class1    1.6127  0.77458   1.0392   1.0392  1.5448   1.547
Join     class1    1.7718        0  0.56524  0.56524  3.0925  1.5608

mode = vector
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    class1    0.64773  0.64773   1.0005   1.0005  0.65557  0.65538

... (15 captured-output lines omitted; rerun the source example for the full output)

JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR    Tput
Delay    class1     1.5688   1.5688   1.0401   1.0401   1.5284  1.5083
Queue1   class1     1.6277  0.79454   1.1405   1.1405   1.5083  1.4271
Queue2   class1    0.61004  0.43665  0.79765  0.79765  0.76875  0.7648
Join     class1          0        0        0        0   2.1407   1.452
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.7 Layered queueing networks

A layered queueing network (LQN) models software resources explicitly. A `Processor` hosts `Tasks`, each task exposes `Entry` points, and each entry runs an `Activity` graph that consumes host demand and makes synchronous or asynchronous calls to the entries of other tasks. A synchronous call blocks the caller until the reply arrives, which is what creates the layers: a task is simultaneously a server to its callers and a client of the tasks it calls.

The models are solved by `SolverLN`, which decomposes the layers into queueing submodels and iterates to a fixed point, and by the external `lqns` and `lqsim` tools. The same model is also readable from, and writable to, the `.lqnx` XML interchange format.

2.7.1 `lqn_basic`

One reference task calling one server task, the smallest layered model.

Processors	2: P1 (ps, $m = 2$), P2 (ps, $m = 3$)
Tasks	3: T1 (ref, $m = 50$), T2 (fcfs, $m = 50$), T3 (fcfs, $m = 25$)
Entries	3: E1, E2, E3
Activities	3, host demands AS1: 0.1, AS2: 0.05, AS3: 0.02
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 2.0

Table 2.54: Features of `lqn_basic`.

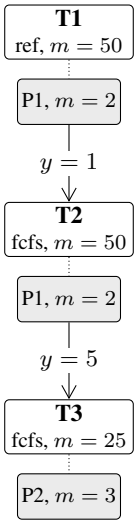

Figure 2.54: Call graph of `lqn_basic`: 3 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

model = LayeredNetwork('test_LQN_4')

P1 = Processor(model, 'P1', 2, SchedStrategy.PS)
P2 = Processor(model, 'P2', 3, SchedStrategy.PS)

T1 = Task(model, 'T1', 50, SchedStrategy.REF).on(P1).set_think_time(Exp(1 / 2))
T2 = Task(model, 'T2', 50, SchedStrategy.FCFS).on(P1).set_think_time(Exp(1 / 3))

T3 = Task(model, 'T3', 25, SchedStrategy.FCFS).on(P2).set_think_time(Exp(1 / 4))

E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T3)

A1 = Activity(model, 'AS1', Exp(10)).on(T1).bound_to(E1).synch_call(E2, 1)
A2 = Activity(model, 'AS2', Exp(20)).on(T2).bound_to(E2).synch_call(E3, 5).replies_to(E2)
A3 = Activity(model, 'AS3', Exp(50)).on(T3).bound_to(E3).replies_to(E3)

avg_table = LN(model).get_avg_table()
print(avg_table)
```

Running this edition prints

```
Node  NodeType     QLen     Util    RespT   ResidT  ArvR  Tput
P1    Processor     NaN  0.99711      NaN      NaN   NaN   NaN
P2    Processor     NaN  0.44316      NaN      NaN   NaN   NaN
T1    RefTask      23.4  0.66474      NaN   1.0848   NaN  13.3
T2    Task       8.7259  0.33237      NaN  0.55607   NaN  13.3
T3    Task       1.3295  0.44316      NaN     0.02   NaN  66.5
E1    Entry        23.4  0.66474   1.7609      NaN   NaN  13.3
E2    Entry      8.7259  0.33237  0.65634      NaN   NaN  13.3
E3    Entry      1.3295  0.44316     0.02      NaN   NaN  66.5
AS1   Activity     23.4  0.66474   1.7609   1.0848   NaN  13.3
AS2   Activity   8.7259  0.33237  0.65634  0.55607   NaN  13.3
AS3   Activity   1.3295  0.44316     0.02     0.02   NaN  66.5
```

2.7.2 `lqn_bpmn`

A BPMN process translated into a layered model with forks, joins and immediate activities.

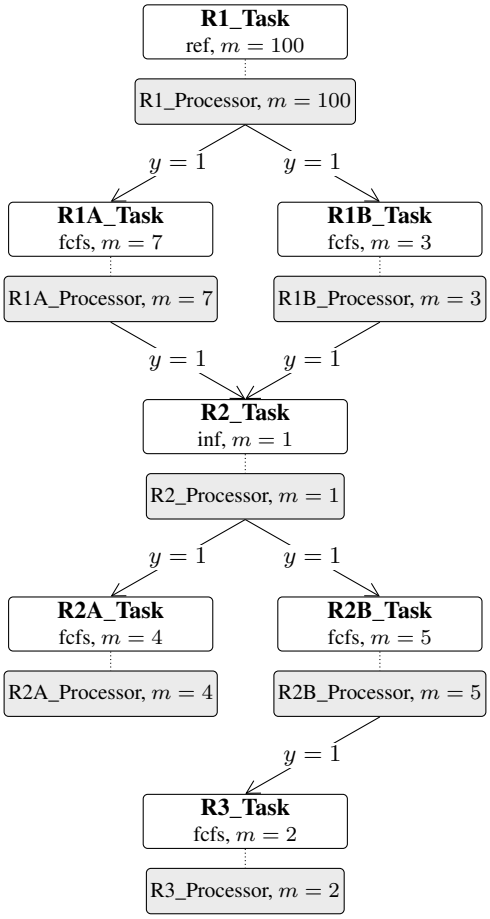

Figure 2.55: Call graph of `lqn_bpmn`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	7: R1_Processor (fcfs, $m = 100$), R2_Processor (inf, $m = 1$), R3_Processor (fcfs, $m = 2$), R1A_Processor (fcfs, $m = 7$), R1B_Processor (fcfs, $m = 3$), R2A_Processor (fcfs, $m = 4$), R2B_Processor (fcfs, $m = 5$)
Tasks	7: R1_Task (ref, $m = 100$), R2_Task (inf, $m = 1$), R3_Task (fcfs, $m = 2$), R1A_Task (fcfs, $m = 7$), R1B_Task (fcfs, $m = 3$), R2A_Task (fcfs, $m = 4$), R2B_Task (fcfs, $m = 5$)
Entries	16: R1_Ref_Entry, R2_Synch_A2_Entry, R2_Synch_A5_Entry, R3_Synch_A9_Entry, R1A_Synch_A1_Entry, R1A_Synch_A2_Entry, R1A_Synch_A3_Entry, R1B_Synch_A4_Entry, R1B_Synch_A5_Entry, R1B_Synch_A6_Entry, R2A_Synch_A7_Entry, R2A_Synch_A8_Entry, R2A_Synch_A11_Entry, R2B_Synch_A9_Entry, R2B_Synch_A10_Entry, R2B_Synch_A12_Entry
Activities	29, host demands A1_Empty: 0.0, A2_Empty: 0.0, A5_Empty: 0.0, A6_Empty: 0.0, A3_Empty: 0.0, A4_Empty: 0.0, E4_Empty: 0.0, A7_Empty: 0.0, ...
Calls	15 (15 synchronous, 0 asynchronous)
Think times	R1_Task: 20.0

Table 2.55: Features of `lqn_bpmn`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# %%
model = LayeredNetwork('myLayeredModel')
# Definition of processors - matching MATLAB exactly
P = []
P.append(Processor(model, 'R1_Processor', 100, SchedStrategy.FCFS))     # P{1}
P.append(Processor(model, 'R2_Processor', float('inf'), SchedStrategy.INF))  # P{2}
P.append(Processor(model, 'R3_Processor', 2, SchedStrategy.FCFS))       # P{3}
P.append(Processor(model, 'R1A_Processor', 7, SchedStrategy.FCFS))      # P{4}
P.append(Processor(model, 'R1B_Processor', 3, SchedStrategy.FCFS))      # P{5}
P.append(Processor(model, 'R2A_Processor', 4, SchedStrategy.FCFS))      # P{6}
P.append(Processor(model, 'R2B_Processor', 5, SchedStrategy.FCFS))      # P{7}
# Definition of tasks - matching MATLAB exactly
T = []
T.append(Task(model, 'R1_Task', 100, SchedStrategy.REF).on(P[0]).set_think_time(Exp.fit_mean
    (20)))  # T{1}
T.append(Task(model, 'R2_Task', float('inf'), SchedStrategy.INF).on(P[1]).set_think_time(
    Immediate()))  # T{2}
T.append(Task(model, 'R3_Task', 2, SchedStrategy.FCFS).on(P[2]).set_think_time(Immediate()))
          # T{3}
T.append(Task(model, 'R1A_Task', 7, SchedStrategy.FCFS).on(P[3]).set_think_time(Immediate()))
          # T{4}
T.append(Task(model, 'R1B_Task', 3, SchedStrategy.FCFS).on(P[4]).set_think_time(Immediate()))
          # T{5}
T.append(Task(model, 'R2A_Task', 4, SchedStrategy.FCFS).on(P[5]).set_think_time(Immediate()))
          # T{6}
T.append(Task(model, 'R2B_Task', 5, SchedStrategy.FCFS).on(P[6]).set_think_time(Immediate()))
          # T{7}
# Definition of entries - matching MATLAB exactly
E = []
E.append(Entry(model, 'R1_Ref_Entry').on(T[0]))          # E{1}
E.append(Entry(model, 'R2_Synch_A2_Entry').on(T[1]))     # E{2}

    ... (78 source lines omitted; full implementation: python/examples/basic/)

options = MVA.default_options()
options.verbose = 0
solverLN = LN(model, lambda model_arg: MVA(model_arg, options), lnoptions)
AvgTableLN = solverLN.avg_table()
print('\nLN(MVA) Results:')
print(AvgTableLN)
```

Running this edition prints

```
This example illustrates the solution of a complex layered queueing network extracted from a BPMN model.

LQNS Results:
Node                NodeType      QLen     Util   RespT  ResidT  ArvR      Tput
R1_Processor        Processor      NaN        0     NaN     NaN   NaN       NaN
R2_Processor        Processor      NaN        0     NaN     NaN   NaN       NaN
R3_Processor        Processor      NaN   1.1895     NaN     NaN   NaN       NaN
R1A_Processor       Processor      NaN   2.0111     NaN     NaN   NaN       NaN
R1B_Processor       Processor      NaN   1.8962     NaN     NaN   NaN       NaN
R2A_Processor       Processor      NaN   1.3324     NaN     NaN   NaN       NaN
R2B_Processor       Processor      NaN   1.7575     NaN     NaN   NaN       NaN
R1_Task             RefTask       98.1        0     NaN     NaN   NaN   0.14365
R2_Task             Task         6.173        0     NaN     NaN   NaN   0.27712
R3_Task             Task        1.2197   1.1895     NaN     NaN   NaN   0.11895
R1A_Task            Task        7.1285   2.0111     NaN     NaN   NaN   0.37349
R1B_Task            Task        3.1688   1.8962     NaN     NaN   NaN   0.34476

... (101 captured-output lines omitted; rerun the source example for the full output)

A5_Res_Empty        Activity    1.3955        0  13.026       0   NaN   0.10713
A7                  Activity   0.69541   0.1532   6.809  2.2697   NaN   0.10213
A8                  Activity   0.92721  0.20426  9.0786  3.0262   NaN   0.10213
A11                 Activity    0.4636  0.10213  4.5393  1.5131   NaN   0.10213
A9                  Activity   0.44037 0.080635   4.369  1.4551   NaN   0.10079
A10                 Activity   0.66055  0.12095  6.5535  2.1826   NaN   0.10079
A12                 Activity   0.88299  0.16168   8.738  2.9176   NaN   0.10105
A9_Res_Empty        Activity    1.2036        0  11.941       0   NaN   0.10079
```

2.7.3 `lqn_bpmn_trace`

The BPMN translation with the activity trace printed.

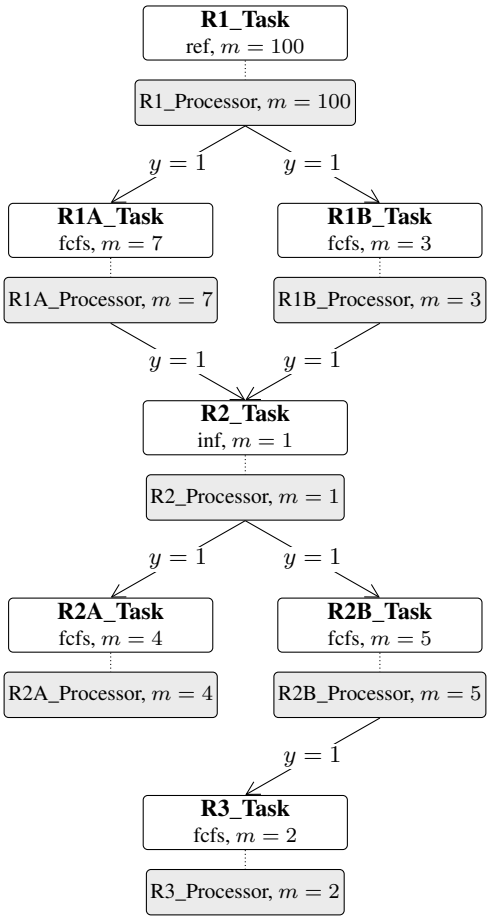

Figure 2.56: Call graph of `lqn_bpmn_trace`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
                    GlobalConstants, Immediate, LayeredNetwork, LN, MVA,
                    Processor, SchedStrategy, Task, VerboseLevel)
model = LayeredNetwork('myLayeredModel')
P = [None] * 8
P[1] = Processor(model, 'R1_Processor', 100, SchedStrategy.FCFS)
P[2] = Processor(model, 'R2_Processor', float('inf'), SchedStrategy.INF)
P[3] = Processor(model, 'R3_Processor', 2, SchedStrategy.FCFS)
P[4] = Processor(model, 'R1A_Processor', 7, SchedStrategy.FCFS)
P[5] = Processor(model, 'R1B_Processor', 3, SchedStrategy.FCFS)
P[6] = Processor(model, 'R2A_Processor', 4, SchedStrategy.FCFS)
P[7] = Processor(model, 'R2B_Processor', 5, SchedStrategy.FCFS)
T = [None] * 8
T[1] = Task(model, 'R1_Task', 100, SchedStrategy.REF).on(P[1]).setThinkTime(Exp.fitMean(20))
T[2] = Task(model, 'R2_Task', float('inf'), SchedStrategy.INF).on(P[2]).setThinkTime(
    Immediate())
T[3] = Task(model, 'R3_Task', 2, SchedStrategy.FCFS).on(P[3]).setThinkTime(Immediate())
T[4] = Task(model, 'R1A_Task', 7, SchedStrategy.FCFS).on(P[4]).setThinkTime(Immediate())
T[5] = Task(model, 'R1B_Task', 3, SchedStrategy.FCFS).on(P[5]).setThinkTime(Immediate())
T[6] = Task(model, 'R2A_Task', 4, SchedStrategy.FCFS).on(P[6]).setThinkTime(Immediate())
```

Processors	7: R1_Processor (fcfs, $m = 100$), R2_Processor (inf, $m = 1$), R3_Processor (fcfs, $m = 2$), R1A_Processor (fcfs, $m = 7$), R1B_Processor (fcfs, $m = 3$), R2A_Processor (fcfs, $m = 4$), R2B_Processor (fcfs, $m = 5$)
Tasks	7: R1_Task (ref, $m = 100$), R2_Task (inf, $m = 1$), R3_Task (fcfs, $m = 2$), R1A_Task (fcfs, $m = 7$), R1B_Task (fcfs, $m = 3$), R2A_Task (fcfs, $m = 4$), R2B_Task (fcfs, $m = 5$)
Entries	16: R1_Ref_Entry, R2_Synch_A2_Entry, R2_Synch_A5_Entry, R3_Synch_A9_Entry, R1A_Synch_A1_Entry, R1A_Synch_A2_Entry, R1A_Synch_A3_Entry, R1B_Synch_A4_Entry, R1B_Synch_A5_Entry, R1B_Synch_A6_Entry, R2A_Synch_A7_Entry, R2A_Synch_A8_Entry, R2A_Synch_A11_Entry, R2B_Synch_A9_Entry, R2B_Synch_A10_Entry, R2B_Synch_A12_Entry
Activities	29, host demands A1_Empty: 0.0, A2_Empty: 0.0, A5_Empty: 0.0, A6_Empty: 0.0, A3_Empty: 0.0, A4_Empty: 0.0, E4_Empty: 0.0, A7_Empty: 0.0, ...
Calls	15 (15 synchronous, 0 asynchronous)
Think times	R1_Task: 20.0

Table 2.56: Features of `lqn_bpmn_trace`.

```
T[7] = Task(model, 'R2B_Task', 5, SchedStrategy.FCFS).on(P[7]).setThinkTime(Immediate())
E = [None] * 17
E[1] = Entry(model, 'R1_Ref_Entry').on(T[1])
E[2] = Entry(model, 'R2_Synch_A2_Entry').on(T[2])
E[3] = Entry(model, 'R2_Synch_A5_Entry').on(T[2])
E[4] = Entry(model, 'R3_Synch_A9_Entry').on(T[3])
E[5] = Entry(model, 'R1A_Synch_A1_Entry').on(T[4])
E[6] = Entry(model, 'R1A_Synch_A2_Entry').on(T[4])

    ... (88 source lines omitted; full implementation: python/examples/basic/)

task_names = ['R1_Task', 'R2_Task', 'R3_Task', 'R1A_Task', 'R1B_Task',
              'R2A_Task', 'R2B_Task']
for name in task_names:
    rows = avg_table[avg_table['Node'].astype(str) == name]
    for tput in np.asarray(rows['Tput'], dtype=float):
        print('%s: Tput=%.6f' % (name, tput))
```

Running this edition prints

```
=== Running LN solver with tracing ===
=== Iteration 1 ===
  Layer 1: max(TN)=5.0000e+00, max(QN)=1.0000e+02
  Layer 2: max(TN)=5.0000e+08, max(QN)=0.0000e+00
  Layer 3: max(TN)=2.0000e-01, max(QN)=2.0000e+00
  Layer 4: max(TN)=3.5000e+08, max(QN)=0.0000e+00
  Layer 5: max(TN)=1.5000e+08, max(QN)=0.0000e+00
  Layer 6: max(TN)=2.0000e+08, max(QN)=0.0000e+00
  Layer 7: max(TN)=2.5000e+08, max(QN)=0.0000e+00
  Layer 8: max(TN)=3.5000e+08, max(QN)=0.0000e+00
  Layer 9: max(TN)=1.8692e+08, max(QN)=0.0000e+00
  Layer 10: max(TN)=5.0000e+00, max(QN)=1.0000e+02
  Layer 11: max(TN)=5.0000e+00, max(QN)=1.0000e+02
  Layer 12: max(TN)=3.8760e+08, max(QN)=0.0000e+00
  Layer 13: max(TN)=4.5042e+08, max(QN)=0.0000e+00
=== Iteration 2 ===

... (55 captured-output lines omitted; rerun the source example for the full output)

=== Final check: Getting AvgTable ===
R1_Task: Tput=0.107222
R2_Task: Tput=0.214269
R3_Task: Tput=0.100166
R1A_Task: Tput=0.277727
R1B_Task: Tput=0.258298
R2A_Task: Tput=0.305607
R2B_Task: Tput=0.302435
```

2.7.4 `lqn_flatph`

Method `flat.ph`: the squashed layering carrying the same composed phase-type encoding, one submodel holding every processor and called task.

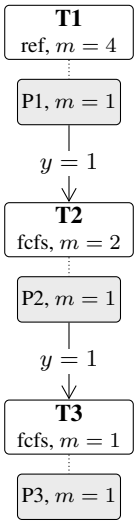

Figure 2.57: Call graph of `lqn_flatph`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	3: P1 (ps, $m = 1$), P2 (ps, $m = 1$), P3 (ps, $m = 1$)
Tasks	3: T1 (ref, $m = 4$), T2 (fcfs, $m = 2$), T3 (fcfs, $m = 1$)
Entries	3: E1, E2, E3
Activities	3, host demands A1: 0.2, A2: 0.2, A3: 0.2
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 1.0

Table 2.57: Features of `lqn_flatph`.

The example is built and solved as follows.

```
                    Processor, SchedStrategy, SolverLN, Task, VerboseLevel)
model = LayeredNetwork('flatphExample')
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
P3 = Processor(model, 'P3', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 4, SchedStrategy.REF).on(P1).setThinkTime(Exp(1))
T2 = Task(model, 'T2', 2, SchedStrategy.FCFS).on(P2)
T3 = Task(model, 'T3', 1, SchedStrategy.FCFS).on(P3)
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T3)
Activity(model, 'A1', Exp(5)).on(T1).boundTo(E1).synchCall(E2, 1)
Activity(model, 'A2', Exp(5)).on(T2).boundTo(E2).synchCall(E3, 1).repliesTo(E2)
Activity(model, 'A3', Exp(5)).on(T3).boundTo(E3).repliesTo(E3)
return model
GlobalConstants.set_verbose(VerboseLevel.SILENT)
model = lqn_flatph()

# The four encodings side by side. 'flat' is the ALIAS of 'flat.cs' and
# resolves unconditionally rather than probing 'flat.ph', because a model is
# squashed in order to express the routed call groups that only the routing
# encoding dispatches; naming 'flat.ph' is therefore not the same as naming
# 'flat'.
for m in ('srvn.cs', 'srvn.ph', 'flat.cs', 'flat.ph'):
    solver = SolverLN(model, method=m, verbose=False)
```

```
    table = solver.getAvgTable()
    print('\n--- method=%s (built as %s, %d submodel(s))'
          % (m, solver.lnmethod, len(solver.ensemble)))
    print(table)
```

Running this edition prints

```
Warning [SolverMVA]: SolverMVA does not handle immediate feedback (immfeed); the solver will treat self-
     loops as class-switching with re-queueing.
[SolverMVA] Message cast more than once, repetitions will not be printed for 60 seconds.

--- method=srvn.cs (built as srvn.cs, 5 submodel(s))
Node  NodeType      QLen     Util    RespT   ResidT  ArvR    Tput
P1    Processor      NaN  0.43679      NaN      NaN   NaN     NaN
P2    Processor      NaN  0.43679      NaN      NaN   NaN     NaN
P3    Processor      NaN  0.43679      NaN      NaN   NaN     NaN
T1    RefTask     1.8161  0.43679      NaN  0.28352   NaN  2.1839
T2    Task        1.0744  0.43679      NaN  0.24599   NaN  2.1839
T3    Task       0.43679  0.43679      NaN      0.2   NaN  2.1839
E1    Entry       1.8161  0.43679  0.83156      NaN   NaN  2.1839
E2    Entry       1.0744  0.43679  0.49198      NaN   NaN  2.1839
E3    Entry      0.43679  0.43679      0.2      NaN   NaN  2.1839
A1    Activity    1.8161  0.43679  0.83156  0.28352   NaN  2.1839
A2    Activity    1.0744  0.43679  0.49198  0.24599   NaN  2.1839

... (38 captured-output lines omitted; rerun the source example for the full output)

T2    Task        1.0665  0.42529      NaN  0.24751   NaN  2.1264
T3    Task       0.42533  0.42533      NaN      0.2   NaN  2.1267
E1    Entry       1.8736  0.42529  0.88108      NaN   NaN  2.1264
E2    Entry       1.0665  0.42529  0.50152      NaN   NaN  2.1264
E3    Entry      0.42533  0.42533      0.2      NaN   NaN  2.1267
A1    Activity    1.8736  0.42529  0.88108  0.28407   NaN  2.1264
A2    Activity    1.0665  0.42529  0.50152  0.24751   NaN  2.1264
A3    Activity   0.42533  0.42533      0.2      0.2   NaN  2.1267
```

2.7.5 `lqn_fork_open_arrival`

An AND-fork under an open arrival stream.

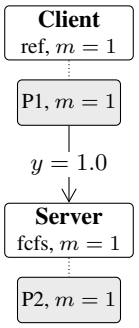

Figure 2.58: Call graph of `lqn_fork_open_arrival`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
model = LayeredNetwork('lqnForkOpenArrival')
P1 = Processor(model, 'P1', float('inf'), SchedStrategy.INF)
T1 = Task(model, 'Client', 1, SchedStrategy.REF).on(P1)
T1.set_think_time(Exp.fit_mean(1.0))
CE = Entry(model, 'CE').on(T1)
P2 = Processor(model, 'P2', float('inf'), SchedStrategy.INF)
T2 = Task(model, 'Server', 1, SchedStrategy.FCFS).on(P2)
T2.set_think_time(Immediate())
SE = Entry(model, 'SE').on(T2)
OE = Entry(model, 'OE').on(T2)
```

Processors	2: P1 (inf, $m = 1$), P2 (inf, $m = 1$)
Tasks	2: Client (ref, $m = 1$), Server (fcfs, $m = 1$)
Entries	3: CE, SE, OE
Activities	9, host demands CA: 0.5, RA1: 0.2, RA2: 0.3, RA3: 0.4, RA4: 0.1, OA1: 0.2, OA2: 0.3, OA3: 0.4, ...
Calls	1 (1 synchronous, 0 asynchronous)
Think times	Client: 1.0

Table 2.58: Features of `lqn_fork_open_arrival`.

```
OE.set_arrival(Exp(0.1))
Activity(model, 'CA', Exp.fit_mean(0.5)).on(T1).bound_to(CE).synch_call(SE)
RA1 = Activity(model, 'RA1', Exp.fit_mean(0.2)).on(T2).bound_to(SE)
RA2 = Activity(model, 'RA2', Exp.fit_mean(0.3)).on(T2)
RA3 = Activity(model, 'RA3', Exp.fit_mean(0.4)).on(T2)
RA4 = Activity(model, 'RA4', Exp.fit_mean(0.1)).on(T2).replies_to(SE)
T2.add_precedence(ActivityPrecedence.AndFork(RA1, [RA2, RA3]))
T2.add_precedence(ActivityPrecedence.AndJoin([RA2, RA3], RA4))
OA1 = Activity(model, 'OA1', Exp.fit_mean(0.2)).on(T2).bound_to(OE)
OA2 = Activity(model, 'OA2', Exp.fit_mean(0.3)).on(T2)
OA3 = Activity(model, 'OA3', Exp.fit_mean(0.4)).on(T2)
OA4 = Activity(model, 'OA4', Exp.fit_mean(0.1)).on(T2)
T2.add_precedence(ActivityPrecedence.AndFork(OA1, [OA2, OA3]))
T2.add_precedence(ActivityPrecedence.AndJoin([OA2, OA3], OA4))
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = lqn_fork_open_arrival()

try:
    print(SolverLN(model, verbose=False).get_avg_table())
except Exception as err:
    print('LN refuses this model: %s' % err)
```

Running this edition prints

```
Node    NodeType   QLen  Util  RespT  ResidT  ArvR  Tput
P1      Processor   NaN   0.2    NaN     NaN   NaN   NaN
P2      Processor   NaN   0.4    NaN     NaN   NaN   NaN
Client  RefTask     0.6   0.2    NaN     0.5   NaN   0.4
Server  Task        0.4   0.4    NaN       1   NaN   0.4
CE      Entry       0.6   0.2    1.5     NaN   NaN   0.4
SE      Entry       0.4   0.4      1     NaN   NaN   0.4
OE      Entry         0     0      1     NaN   NaN     0
CA      Activity    0.6   0.2    1.5     0.5   NaN   0.4
RA1     Activity   0.08  0.08    0.2     0.2   NaN   0.4
RA2     Activity   0.12  0.12    0.3     0.3   NaN   0.4
RA3     Activity   0.16  0.16    0.4     0.4   NaN   0.4
RA4     Activity   0.04  0.04    0.1     0.1   NaN   0.4
OA1     Activity      0     0    0.2       0   NaN     0
OA2     Activity      0     0    0.3       0   NaN     0
OA3     Activity      0     0    0.4       0   NaN     0
OA4     Activity      0     0    0.1       0   NaN     0
```

2.7.6 `lqn_jsq`

Join-the-shortest-queue dispatching between replicated tasks.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
model = LayeredNetwork('LQN-JSQ')
PC = Processor(model, 'PC', 1, SchedStrategy.INF)
PS = Processor(model, 'PS', 1, SchedStrategy.PS)
TC = Task(model, 'TC', 10, SchedStrategy.REF).on(PC).set_think_time(Exp(1 / 5))
```

```
TS1 = Task(model, 'TS1', 5, SchedStrategy.FCFS).on(PS)
TS2 = Task(model, 'TS2', 5, SchedStrategy.FCFS).on(PS)
TS3 = Task(model, 'TS3', 5, SchedStrategy.FCFS).on(PS)
EC = Entry(model, 'EC').on(TC)
ES1 = Entry(model, 'ES1').on(TS1)
ES2 = Entry(model, 'ES2').on(TS2)
ES3 = Entry(model, 'ES3').on(TS3)
# ONE call per invocation, its destination the least loaded of the three
# servers. As in the round-robin twin, mean_calls=1 over 3 targets is where
# the strategy bites: the probabilistic model makes 0..3 calls per
# invocation with the same mean, JSQ makes exactly one, to the shortest
# queue.
Activity(model, 'AC', Exp(2)).on(TC).bound_to(EC).synch_call_jsq([ES1, ES2, ES3], 1)
Activity(model, 'AS1', Exp(1)).on(TS1).bound_to(ES1).replies_to(ES1)
Activity(model, 'AS2', Exp(1)).on(TS2).bound_to(ES2).replies_to(ES2)
Activity(model, 'AS3', Exp(1)).on(TS3).bound_to(ES3).replies_to(ES3)
return model
"""Servers of unequal speed, where JSQ and round-robin part company.
Round-robin hands each server exactly one third of the calls whatever their
speed; JSQ follows the queues, so the fast server absorbs more of them and
the entry throughputs come out unequal.
"""

    ... (13 source lines omitted; full implementation: python/examples/basic/)

GlobalConstants.set_verbose(VerboseLevel.SILENT)

options = SolverLNOptions()
options.config['layering'] = 'flat'
options.verbose = False
print(SolverLN(lqn_jsq(), SolverSSA, options).get_avg_table())
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.7.7 `lqn_moment3`

The third moment of the entry service time, and how it propagates up the layers.

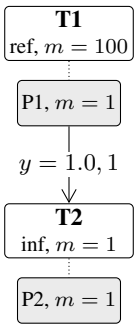

Figure 2.59: Call graph of `lqn_moment3`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
model = LayeredNetwork('myLayeredModel')
# Definition of processors, tasks and entries
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 100, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T2 = Task(model, 'T2', 1, SchedStrategy.INF).on(P2)
E2 = Entry(model, 'E2').on(T2)
```

Processors	2: P1 (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	2: T1 (ref, $m = 100$), T2 (inf, $m = 1$)
Entries	3: E1, E2, E3
Activities	5, host demands A1: 1.0, A20: 1.0, A21: 1.0, A22: 1.0, A3: 1.0
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 10.0
Solvers	FLD

Table 2.59: Features of `lqn_moment3`.

```
E3 = Entry(model, 'E3').on(T2)
# Definition of activities
T1.set_think_time(Erlang.fit_mean_and_order(10, 1))
A1 = Activity(model, 'A1', Exp(1)).on(T1).bound_to(E1).synch_call(E2).synch_call(E3, 1)
A20 = Activity(model, 'A20', Exp(1)).on(T2).bound_to(E2)
A21 = Activity(model, 'A21', Exp(1)).on(T2)
A22 = Activity(model, 'A22', Exp(1)).on(T2).replies_to(E2)
T2.add_precedence(ActivityPrecedence.serial([A20, A21, A22]))
A3 = Activity(model, 'A3', Exp(1)).on(T2).bound_to(E3).replies_to(E3)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = lqn_moment3()

# LN solver with default method (using NC layer solver)
solver_ln = LN(model, lambda m: NC(m, verbose=False), verbose=False)
avg_table_ln = solver_ln.get_avg_table()
print('\nLN Results:')
print(avg_table_ln)

    ... (9 source lines omitted; full implementation: python/examples/basic/)

    print(avg_table_ln2)
except Exception as e:
    import sys
    print(f'\nLN(moment3) failed: {e}', file=sys.stderr)
    print('\nLN(moment3) Results:')
    print(avg_table_ln)
```

Running this edition prints

```
LN Results:
Node   NodeType    QLen  Util  RespT  ResidT  ArvR  Tput
P1     Processor    NaN  0.25    NaN     NaN   NaN   NaN
P2     Processor    NaN     1    NaN     NaN   NaN   NaN
T1     RefTask     97.5  0.25    NaN  1.3275   NaN  0.25
T2     Task        97.2     1    NaN   194.3   NaN   0.5
E1     Entry       97.5  0.25    390     NaN   NaN  0.25
E2     Entry       72.9  0.75  291.5     NaN   NaN  0.25
E3     Entry       24.3  0.25   97.2     NaN   NaN  0.25
A1     Activity    97.5  0.25    390  1.3275   NaN  0.25
A20    Activity    24.3  0.25   97.2    48.6   NaN  0.25
A21    Activity    24.3  0.25   97.2    48.6   NaN  0.25
A22    Activity    24.3  0.25   97.2    48.6   NaN  0.25
A3     Activity    24.3  0.25   97.2    48.6   NaN  0.25
Warning [SolverNC]: SolverNC does not handle immediate feedback (immfeed); the solver will treat self-
    loops as class-switching with re-queueing.
[SolverNC] Message cast more than once, repetitions will not be printed for 60 seconds.

... (15 captured-output lines omitted; rerun the source example for the full output)

E1     Entry      0.26874  0.25  1.075     NaN   NaN  0.25
E2     Entry         72.9  0.75  291.5     NaN   NaN  0.25
E3     Entry         24.3  0.25   97.2     NaN   NaN  0.25
A1     Activity      97.5  0.25    390  1.3275   NaN  0.25
A20    Activity      24.3  0.25   97.2    48.6   NaN  0.25
A21    Activity      24.3  0.25   97.2    48.6   NaN  0.25
A22    Activity      24.3  0.25   97.2    48.6   NaN  0.25
A3     Activity      24.3  0.25   97.2    48.6   NaN  0.25
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and

throughput of each station-class pair, as returned by `FLD`.

2.7.8 `lqn_multi_solvers`

One layered model solved by every layered solver, side by side.

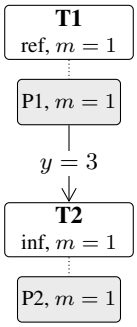

Figure 2.60: Call graph of `lqn_multi_solvers`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (inf, $m = 1$), P2 (inf, $m = 1$)
Tasks	2: T1 (ref, $m = 1$), T2 (inf, $m = 1$)
Entries	2: E1, E2
Activities	2, host demands A1: 1.0, A2: 1.0
Calls	1 (1 synchronous, 0 asynchronous)
Think times	T1: 0.0001

Table 2.60: Features of `lqn_multi_solvers`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# Clear variables (equivalent to MATLAB's clear solver AvgTable)
solver = None
AvgTable = None
# %%
# MATLAB: model = LayeredNetwork('LQN1');
model = LayeredNetwork('LQN1')
# definition of processors, tasks and entries
# MATLAB: P1 = Processor(model, 'P1', Inf, SchedStrategy.INF);
# MATLAB: T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1);
# MATLAB: E1 = Entry(model, 'E1').on(T1);
P1 = Processor(model, 'P1', float('inf'), SchedStrategy.INF)
T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
# MATLAB: P2 = Processor(model, 'P2', Inf, SchedStrategy.INF);
# MATLAB: T2 = Task(model, 'T2', Inf, SchedStrategy.INF).on(P2);
# MATLAB: E2 = Entry(model, 'E2').on(T2);
P2 = Processor(model, 'P2', float('inf'), SchedStrategy.INF)
T2 = Task(model, 'T2', float('inf'), SchedStrategy.INF).on(P2)
E2 = Entry(model, 'E2').on(T2)
# definition of activities
# MATLAB: T1.set_think_time(Erlang.fit_mean_and_order(0.0001,2));
T1.set_think_time(Erlang.fit_mean_and_order(0.0001, 2))
# MATLAB: A1 = Activity(model, 'A1', Exp(1.0)).on(T1).boundTo(E1).synchCall(E2,3);
# MATLAB: A2 = Activity(model, 'A2', APH.fit_mean_and_scv(1,10)).on(T2).boundTo(E2).repliesTo
    (E2);

    ... (64 source lines omitted; full implementation: python/examples/basic/)

# MATLAB: %AvgTable{2} = solver{2}.get_avg_table
```

```
# MATLAB: %AvgTable{2}
# solver[3] = LN(model, lambda model_arg: LINE(model_arg, LINE.default_options()), lnoptions)
# AvgTable[3] = solver[3].avg_table()
# print('AvgTable[3]:')
# print(AvgTable[3])
```

Running this edition prints

```
LQNS command: lqns -Pmultiserver=rolia -Pstop-on-message-loss=false -x <tmp>
AvgTableLQNS:
Node  NodeType      QLen     Util  RespT  ResidT  ArvR     Tput
P1    Processor      NaN  0.24999    NaN     NaN   NaN      NaN
P2    Processor      NaN  0.74998    NaN     NaN   NaN      NaN
T1    RefTask          1  0.24999    NaN     NaN   NaN  0.24999
T2    Task       0.74998  0.74998    NaN     NaN   NaN  0.74998
E1    Entry            1  0.24999      4     NaN   NaN  0.24999
E2    Entry      0.74998  0.74998      1     NaN   NaN  0.74998
A1    Activity         1  0.24999      4     NaN   NaN  0.24999
A2    Activity   0.74998  0.74998      1     NaN   NaN  0.74998
AvgTable[1]:
Node  NodeType      QLen     Util  RespT  ResidT  ArvR     Tput
P1    Processor      NaN  0.24999    NaN     NaN   NaN      NaN
P2    Processor      NaN  0.74998    NaN     NaN   NaN      NaN
T1    RefTask          1  0.24999    NaN       1   NaN  0.24999

... (7 captured-output lines omitted; rerun the source example for the full output)

P1    Processor      NaN  0.24999    NaN     NaN   NaN      NaN
P2    Processor      NaN  0.74998    NaN     NaN   NaN      NaN
T1    RefTask          1  0.24999    NaN       1   NaN  0.24999
T2    Task       0.74998  0.74998    NaN       1   NaN  0.74998
E1    Entry            1  0.24999      4     NaN   NaN  0.24999
E2    Entry      0.74998  0.74998      1     NaN   NaN  0.74998
A1    Activity         1  0.24999      4       1   NaN  0.24999
A2    Activity   0.74998  0.74998      1       1   NaN  0.74998
```

2.7.9 `lqn_ofbiz`

The Apache OFBiz case study, read from its `.lqnx` file.
The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# This example is temporarily disabled (matching MATLAB comment)
# Complex XML parsing may have incomplete support
# Clear variables (equivalent to MATLAB's clear solver AvgTable)
solver = None
AvgTable = None
# %%
print('This example illustrates the solution of a moderately large LQN.')
# MATLAB: cwd = fileparts(which(mfilename));
# MATLAB: model = LayeredNetwork.parseXML([cwd,filesep,'ofbizExample.xml']);
cwd = os.path.dirname(os.path.abspath(__file__)) if '__file__' in globals() else os.getcwd()
xml_path = os.path.join(cwd, 'lqn_ofbiz.xml')
#xml_path = os.path.join(cwd, 'lqn_ofbiz_fcfs.xml')
try:
    # Parse XML file like MATLAB: model = LayeredNetwork.parse_xml(filename)
    model = LayeredNetwork.parse_xml(xml_path)
    print(f'Successfully loaded model from {xml_path}')
except Exception as e:
    print(f'XML parsing failed: {e}')
    model = None
# %%
if model is None:
    print('Model could not be loaded, skipping solver execution')
    exit(0)
# MATLAB: options = LQNS.defaultOptions;

    ... (32 source lines omitted; full implementation: python/examples/basic/)

    Tnoinit = time.time() - Tnoinit_start
```

Processors	9: FrontEnd_CPU_Processor (ps, $m = 1$), USAGE_DELAY_Processor (inf, $m = 1$), UsageScenario_userType1_1_Processor (fcfs, $m = 1$), RequestHandler_HandlerIF_main_345_Processor (fcfs, $m = 1$), RequestHandler_HandlerIF_login_345_Processor (fcfs, $m = 1$), RequestHandler_HandlerIF_checkLogin_345_Processor (fcfs, $m = 1$), RequestHandler_HandlerIF_logout_345_Processor (fcfs, $m = 1$), UsageScenario_userType2_7_Processor (fcfs, $m = 1$), RequestHandler_HandlerIF_quickadd_345_Processor (fcfs, $m = 1$)
Tasks	9: FrontEnd_CPU_Task (inf, $m = 1$), USAGE_DELAY_Task (inf, $m = 1$), UsageScenario_userType1_1_Task (ref, $m = 10$), RequestHandler_HandlerIF_main_345_Task (inf, $m = 1$), RequestHandler_HandlerIF_login_345_Task (inf, $m = 1$), RequestHandler_HandlerIF_checkLogin_345_Task (inf, $m = 1$), RequestHandler_HandlerIF_logout_345_Task (inf, $m = 1$), UsageScenario_userType2_7_Task (ref, $m = 10$), RequestHandler_HandlerIF_quickadd_345_Task (inf, $m = 1$)
Entries	14: FrontEnd_CPU_Entry, InternalAction_main__Iu1-wMhoEeKON4DtRoKCMw_34_50_Entry, InternalAction_login__YgxRGw26EeSPwb7XgvxhWQ_34_50_Entry, InternalAction_CheckLogin__vHMZsMhoEeKON4DtRoKCMw_34_50_Entry, InternalAction_logout__j-2E4MhrEeKON4DtRoKCMw_34_50_Entry, InternalAction_addcart__5JEHQMhoEeKON4DtRoKCMw_34_50_Entry, USAGE_DELAY0_Entry, UsageScenario_userType1_1_Entry, RequestHandler_HandlerIF_main_345_Entry, RequestHandler_HandlerIF_login_345_Entry, RequestHandler_HandlerIF_checkLogin_345_Entry, RequestHandler_HandlerIF_logout_345_Entry, UsageScenario_userType2_7_Entry, RequestHandler_HandlerIF_quickadd_345_Entry
Activities	40, host demands FrontEnd_CPU_Activity: 0.0, InternalAction_main__Iu1-wMhoEeKON4DtRoKCMw_34_50_Activity: 0.01, InternalAction_login__YgxRGw26EeSPwb7XgvxhWQ_34_50_Activity: 0.01, InternalAction_CheckLogin__vHMZsMhoEeKON4DtRoKCMw_34_50_Activity: 0.01, InternalAction_logout__j-2E4MhrEeKON4DtRoKCMw_34_50_Activity: 0.01, InternalAction_addcart__5JEHQMhoEeKON4DtRoKCMw_34_50_Activity: 0.01, USAGE_DELAY0_Activity: 0.0, Start2: 0.0, ...
Calls	19 (19 synchronous, 0 asynchronous)
Think times	UsageScenario_userType1_1_Task: 10.0, UsageScenario_userType2_7_Task: 14.0

Table 2.61: Features of `lqn_ofbiz`.

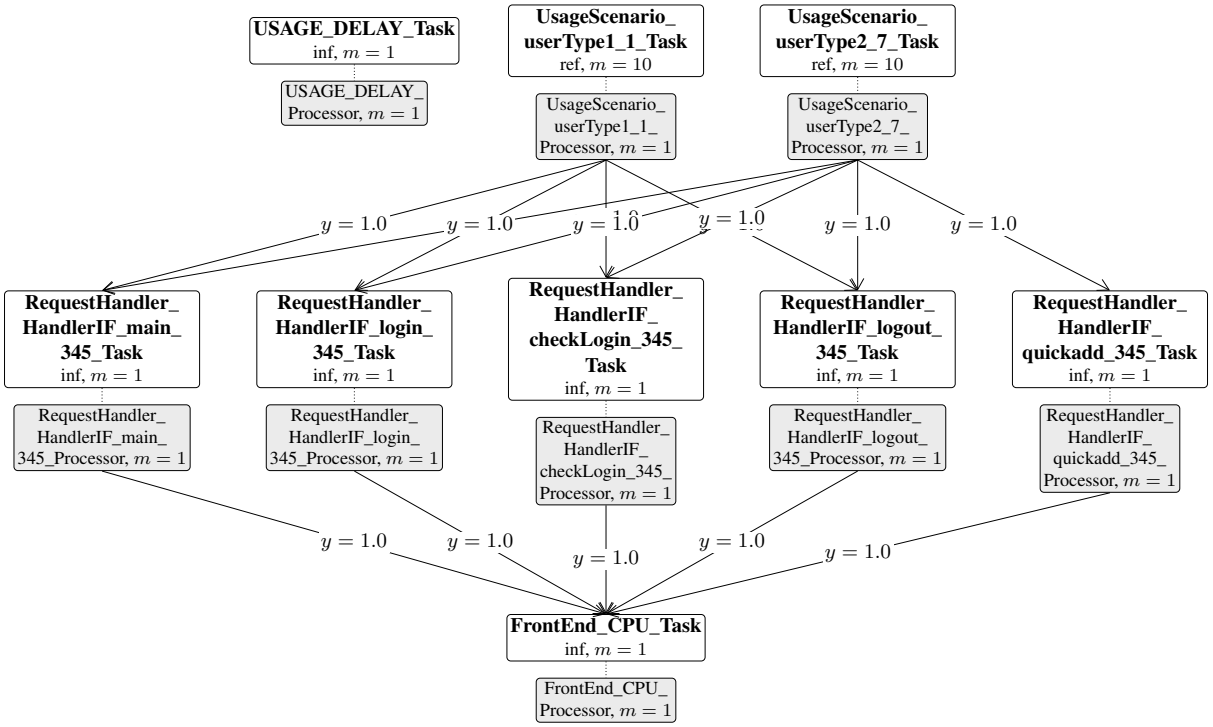

Figure 2.61: Call graph of `lqn_ofbiz`: 9 tasks over 9 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
    print('\nLN(NC) Results:')
    print(AvgTable[2])
    print(f'Tnoinit = {Tnoinit:.6f}')
except Exception as e:
    print(f'LN(NC) solver failed: {e}')
```

Running this edition prints

```
This example illustrates the solution of a moderately large LQN.
Successfully loaded model from python/examples/basic/layeredModel/lqn_ofbiz.xml

LN(NC) Results:
Node                                                    NodeType        QLen       Util      RespT
           ResidT  ArvR     Tput
FrontEnd_CPU_Processor                                  Processor        NaN    0.11637        NaN
              NaN   NaN      NaN
USAGE_DELAY_Processor                                   Processor        NaN          0        NaN
              NaN   NaN      NaN
UsageScenario_userType1_1_Processor                     Processor        NaN          0        NaN
              NaN   NaN      NaN
RequestHandler_HandlerIF_main_345_Processor             Processor        NaN          0        NaN
              NaN   NaN      NaN
RequestHandler_HandlerIF_login_345_Processor            Processor        NaN          0        NaN
              NaN   NaN      NaN
RequestHandler_HandlerIF_checkLogin_345_Processor       Processor        NaN          0        NaN
              NaN   NaN      NaN
RequestHandler_HandlerIF_logout_345_Processor           Processor        NaN          0        NaN
              NaN   NaN      NaN
UsageScenario_userType2_7_Processor                     Processor        NaN          0        NaN
              NaN   NaN      NaN
RequestHandler_HandlerIF_quickadd_345_Processor         Processor        NaN          0        NaN
              NaN   NaN      NaN
FrontEnd_CPU_Task                                       Task         0.13148    0.11637        NaN
         0.011298   NaN   11.637
USAGE_DELAY_Task                                        Task               0          0        NaN
                0   NaN        0

... (54 captured-output lines omitted; rerun the source example for the full output)
```

```
EntryLevelSystemCallcall2quickadd                                Activity   0.0080185          0  0.011298
                0    NaN   0.7097
EntryLevelSystemCallcall2logout                                  Activity   0.0080185          0  0.011298
                0    NaN   0.7097
EntryLevelSystemCallcall2main3                                   Activity   0.0080185          0  0.011298
                0    NaN   0.7097
Stop9                                                            Activity           0          0         0
                0    NaN   0.7097
StartAction_start__EkBkIMhoEeKON4DtRoKCMw_34_5                   Activity           0          0         0
                0    NaN   0.7097
InternalAction_addcart__5JEHQMhoEeKON4DtRoKCMw_34_50             Activity   0.0080185          0  0.011298
                0    NaN   0.7097
StopAction_stop__EkCLMMhoEeKON4DtRoKCMw_34_5                     Activity           0          0         0
                0    NaN   0.7097
Tnoinit = 9.472727
```

2.7.10 `lqn_open_arrival`

An open arrival stream into an entry, instead of a closed reference task.

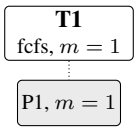

Figure 2.62: Call graph of `lqn_open_arrival`: 1 task over 1 processor, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	1: P1 (ps, $m = 1$)
Tasks	1: T1 (fcfs, $m = 1$)
Entries	1: E1
Activities	1, host demands A1: 1.6

Table 2.62: Features of `lqn_open_arrival`.

The example is built and solved as follows.

```
model = LayeredNetwork('openArrivalLQN')
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 1, SchedStrategy.FCFS).on(P1)
T1.set_think_time(Immediate())
E1 = Entry(model, 'E1').on(T1)
E1.setArrival(Exp(0.2))
Activity(model, 'A1', Exp.fit_mean(1.6)).on(T1).bound_to(E1).replies_to(E1)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = lqn_open_arrival()

avg_table_ln = LN(model).get_avg_table()
print('\nLN Results:')
print(avg_table_ln)
```

Running this edition prints

```
LN Results:
Node  NodeType   QLen  Util  RespT  ResidT  ArvR  Tput
P1    Processor   NaN  0.32    NaN     NaN   NaN   NaN
T1    Task       0.32  0.32    NaN     1.6   NaN   0.2
E1    Entry      0.32  0.32    1.6     NaN   NaN   0.2
A1    Activity   0.32  0.32    1.6     1.6   NaN   0.2
```

2.7.11 lqn_paramident

Parameter identification on a layered model, fitting host demands to measurements.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# light populations keep the native SolverLN loop fast
model = LayeredNetwork('paramident_LQN')
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 5, SchedStrategy.REF).on(P1)
T1.set_think_time(Exp.fit_mean(0.5))
T2 = Task(model, 'T2', 5, SchedStrategy.FCFS).on(P1)
T2.set_think_time(Exp.fit_mean(1.0 / 3.0))
T3 = Task(model, 'T3', 3, SchedStrategy.FCFS).on(P2)
T3.set_think_time(Exp.fit_mean(0.25))
Entry(model, 'E1').on(T1)
Entry(model, 'E2').on(T2)
Entry(model, 'E3').on(T3)
E1, E2, E3 = model.entries
Activity(model, 'AS1', Exp.fit_mean(0.1)).on(T1).bound_to(E1).synch_call(E2, 1)
Activity(model, 'AS2', Exp.fit_mean(0.05)).on(T2).bound_to(E2).synch_call(E3, 5).replies_to(
    E2)
Activity(model, 'AS3', Exp.fit_mean(0.02)).on(T3).bound_to(E3).replies_to(E3)
return model
np.random.seed(12345)
model = build_lqn()
param_spec = [{'type': 'think', 'name': 'T1'},
              {'type': 'hostdem', 'name': 'AS3'}]
obs_spec = [{'metric': 'RespT', 'name': 'E1'},
            {'metric': 'Util', 'name': 'P1'},
            {'metric': 'Util', 'name': 'P2'}]
nsteps = 16

    ... (21 source lines omitted; full implementation: python/examples/basic/)

        a_true_seq[1, k], info['ahat'][1, k],
        float(np.linalg.norm(info['e'][:, k]))))
print('\nFinal: Z = %.4f (true %.4f), Sd = %.5f (true %.5f)' % (
    info['ahat'][0, -1], a_true_seq[0, -1], info['ahat'][1, -1], a_true_seq[1, -1]))
print('RMS tracking Ea = %.4g, prediction Er = %.4g' % (info['Ea'], info['Er']))
main()
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

2.7.12 lqn_rrobin

Round-robin dispatching between replicated tasks.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
model = LayeredNetwork('LQN-RRobin')
PC = Processor(model, 'PC', 1, SchedStrategy.INF)
PS = Processor(model, 'PS', 1, SchedStrategy.PS)
TC = Task(model, 'TC', 10, SchedStrategy.REF).on(PC).set_think_time(Exp(1 / 5))
TS1 = Task(model, 'TS1', 5, SchedStrategy.FCFS).on(PS)
TS2 = Task(model, 'TS2', 5, SchedStrategy.FCFS).on(PS)
TS3 = Task(model, 'TS3', 5, SchedStrategy.FCFS).on(PS)
EC = Entry(model, 'EC').on(TC)
ES1 = Entry(model, 'ES1').on(TS1)
```

```
ES2 = Entry(model, 'ES2').on(TS2)
ES3 = Entry(model, 'ES3').on(TS3)
# ONE call per invocation, its destination cycling over the three servers.
# mean_calls=1 over 3 targets is where round-robin actually bites: the
# probabilistic twin makes 0..3 calls per invocation with the same mean,
# round-robin makes exactly one. At mean_calls=3 the two agree on
# everything except the order.
Activity(model, 'AC', Exp(2)).on(TC).bound_to(EC).synch_call_rrobin([ES1, ES2, ES3], 1)
Activity(model, 'AS1', Exp(1)).on(TS1).bound_to(ES1).replies_to(ES1)
Activity(model, 'AS2', Exp(1)).on(TS2).bound_to(ES2).replies_to(ES2)
Activity(model, 'AS3', Exp(1)).on(TS3).bound_to(ES3).replies_to(ES3)
return model
"""The ungrouped twin: same per-target call means, branching at random."""
model = LayeredNetwork('LQN-RRobin-Prob')
PC = Processor(model, 'PC', 1, SchedStrategy.INF)
PS = Processor(model, 'PS', 1, SchedStrategy.PS)
TC = Task(model, 'TC', 10, SchedStrategy.REF).on(PC).set_think_time(Exp(1 / 5))

    ... (16 source lines omitted; full implementation: python/examples/basic/)

options.config['layering'] = 'flat'
options.verbose = False
# SolverMVA is rejected here: a routed call group needs a layer solver with
# state-dependent routing, and MVA would silently return the probabilistic
# split under a round-robin label.
print(SolverLN(lqn_rrobin(), SolverSSA, options).get_avg_table())
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.7.13 `lqn_serial`

A serial call chain across three layers, with an immediate activity.

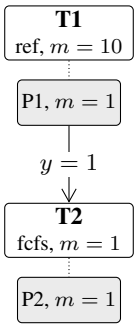

Figure 2.63: Call graph of `lqn_serial`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	2: T1 (ref, $m = 10$), T2 (fcfs, $m = 1$)
Entries	2: E1, E2
Activities	4, host demands AS1: 1.6, AS2: 0.0, AS3: 5.0, AS4: 1.0
Calls	1 (1 synchronous, 0 asynchronous)
Think times	T1: 100.0

Table 2.63: Features of `lqn_serial`.

The example is built and solved as follows.

```
model = LayeredNetwork('myLayeredModel')
```

```
P = np.empty(2, dtype=object)
P[0] = Processor(model, 'P1', 1, SchedStrategy.PS)
P[1] = Processor(model, 'P2', 1, SchedStrategy.PS)
T = np.empty(2, dtype=object)
T[0] = Task(model, 'T1', 10, SchedStrategy.REF).on(P[0]).set_think_time(Exp.fit_mean(100))
T[1] = Task(model, 'T2', 1, SchedStrategy.FCFS).on(P[1]).set_think_time(Immediate())
E = np.empty(2, dtype=object)
E[0] = Entry(model, 'E1').on(T[0])
E[1] = Entry(model, 'E2').on(T[1])
A = np.empty(4, dtype=object)
A[0] = Activity(model, 'AS1', Exp.fit_mean(1.6)).on(T[0]).bound_to(E[0])
A[1] = Activity(model, 'AS2', Immediate()).on(T[0]).synch_call(E[1], 1)
A[2] = Activity(model, 'AS3', Exp.fit_mean(5)).on(T[1]).bound_to(E[1])
A[3] = Activity(model, 'AS4', Exp.fit_mean(1)).on(T[1]).replies_to(E[1])
# Serial precedence for activities
T[0].add_precedence(ActivityPrecedence.serial([A[0], A[1]]))
T[1].add_precedence(ActivityPrecedence.serial([A[2], A[3]]))
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = lqn_serial()

# Solve using LQNS
solver = LQNS(model, keep=True)
avg_table = solver.get_avg_table()
print('\nLQNS Results:')

    ... (7 source lines omitted; full implementation: python/examples/basic/)

# print(call_avg_table)

# Solve using LN
avg_table_ln = LN(model).get_avg_table()
print('\nLN Results:')
print(avg_table_ln)
```

Running this edition prints

```
LQNS command: lqns -Pmultiserver=rolia -Pstop-on-message-loss=false -x <tmp>

LQNS Results:
Node  NodeType       QLen      Util   RespT  ResidT  ArvR      Tput
P1    Processor       NaN   0.14206     NaN     NaN   NaN       NaN
P2    Processor       NaN   0.53271     NaN     NaN   NaN       NaN
T1    RefTask      1.1215   0.14206     NaN     NaN   NaN  0.088786
T2    Task        0.53271   0.53271     NaN     NaN   NaN  0.088786
E1    Entry        1.1215   0.14206  12.631     NaN   NaN  0.088786
E2    Entry       0.53271   0.53271       6     NaN   NaN  0.088786
AS1   Activity    0.16271   0.14206  1.8327     NaN   NaN  0.088786
AS2   Activity    0.95874         0    10.8     NaN   NaN  0.088786
AS3   Activity    0.44393   0.44393       5     NaN   NaN  0.088786
AS4   Activity   0.088786  0.088786       1     NaN   NaN  0.088786

LN Results:

... (3 captured-output lines omitted; rerun the source example for the full output)

T1    RefTask      1.1733   0.14123     NaN  1.8289   NaN  0.088267
T2    Task         0.5296    0.5296     NaN       6   NaN  0.088267
E1    Entry        1.1733   0.14123    13.3     NaN   NaN  0.088267
E2    Entry        0.5296    0.5296       6     NaN   NaN  0.088267
AS1   Activity    0.16143   0.14123  1.8289  1.8289   NaN  0.088267
AS2   Activity     1.0119         0  11.464       0   NaN  0.088267
AS3   Activity    0.44133   0.44133       5       5   NaN  0.088267
AS4   Activity   0.088267  0.088267       1       1   NaN  0.088267
```

2.7.14 `lqn_server_pools`

A layered model with heterogeneous server pools, compared with the compatibility representation that treats the servers as interchangeable.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to

draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
P1 (3 servers)      S1 --- T2
                    S2 --< T2, T3
                    S3 --- T3
Dorsman & Gardner (2024). New directions in pass-and-swap queues.
Queueing Systems 107(3):205-256, Fig. 1.
                     Exp, SchedStrategy, LN, MVA)
model = LayeredNetwork('LQNserverPools')
P0 = Processor(model, 'P0', 1, SchedStrategy.PS)
P1 = Processor(model, 'P1', 3, SchedStrategy.PS)
T1 = Task(model, 'T1', 3, SchedStrategy.REF).on(P0).setThinkTime(Exp(1.0))
T2 = Task(model, 'T2', 3, SchedStrategy.FCFS).on(P1)
T3 = Task(model, 'T3', 3, SchedStrategy.FCFS).on(P1)
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T3)
Activity(model, 'A1', Exp(2.0)).on(T1).boundTo(E1).synchCall(E2, 1).synchCall(E3, 1)
Activity(model, 'A2', Exp(3.0)).on(T2).boundTo(E2).repliesTo(E2)
Activity(model, 'A3', Exp(2.0)).on(T3).boundTo(E3).repliesTo(E3)
if kind == 'homogeneous':
    # one pool of three, every task eligible on every server: the neutral
    # declaration, which reproduces the plain multiserver
    P1.addServerType(ServerType('All', 3, [T2, T3]))
else:
    P1.addServerType(ServerType('S1', 1, [T2]))        # dedicated to T2
    P1.addServerType(ServerType('S2', 1, [T2, T3]))    # shared
    P1.addServerType(ServerType('S3', 1, [T3]))        # dedicated to T3
return model
for kind in ('homogeneous', 'compatibility'):
    print('\n--- %s pool on P1 ---' % kind)
    print(LN(build(kind), lambda m: MVA(m), method='srvn.cs').getAvgTable())
lqn_server_pools()
```

Running this edition prints

```
--- homogeneous pool on P1 ---
Warning [SolverMVA]: SolverMVA does not handle immediate feedback (immfeed); the solver will treat self-
     loops as class-switching with re-queueing.
[SolverMVA] Message cast more than once, repetitions will not be printed for 60 seconds.
Node  NodeType      QLen     Util    RespT   ResidT  ArvR    Tput
P0    Processor      NaN  0.67889      NaN      NaN   NaN     NaN
P1    Processor      NaN   1.1315      NaN      NaN   NaN     NaN
T1    RefTask     1.6422  0.67889      NaN  0.81118   NaN  1.3578
T2    Task       0.22281   0.4526      NaN   0.1641   NaN  1.3578
T3    Task       0.31588  0.67889      NaN  0.23264   NaN  1.3578
E1    Entry       1.6422  0.67889   1.2095      NaN   NaN  1.3578
E2    Entry      0.22281   0.4526   0.1641      NaN   NaN  1.3578
E3    Entry      0.31588  0.67889  0.23264      NaN   NaN  1.3578
A1    Activity    1.6422  0.67889   1.2095  0.81118   NaN  1.3578
A2    Activity   0.22281   0.4526   0.1641   0.1641   NaN  1.3578
A3    Activity   0.31588  0.67889  0.23264  0.23264   NaN  1.3578


... (5 captured-output lines omitted; rerun the source example for the full output)

T2    Task       0.28842  0.42621      NaN  0.22557   NaN  1.2786
T3    Task       0.42476  0.63931      NaN   0.3322   NaN  1.2786
E1    Entry       1.7214  0.63931   1.3463      NaN   NaN  1.2786
E2    Entry      0.28842  0.42621  0.22557      NaN   NaN  1.2786
E3    Entry      0.42476  0.63931   0.3322      NaN   NaN  1.2786
A1    Activity    1.7214  0.63931   1.3463   0.7846   NaN  1.2786
A2    Activity   0.28842  0.42621  0.22557  0.22557   NaN  1.2786
A3    Activity   0.42476  0.63931   0.3322   0.3322   NaN  1.2786
```

2.7.15 `lqn_setup`

Task setup and initialization activities before service.

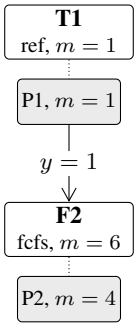

Figure 2.64: Call graph of `lqn_setup`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (inf, $m = 1$), P2 (fcfs, $m = 4$)
Tasks	2: T1 (ref, $m = 1$), F2 (fcfs, $m = 6$)
Entries	2: E1, E2
Activities	2, host demands A1: 1.0, A2: 0.3333333333333333
Calls	1 (1 synchronous, 0 asynchronous)

Table 2.64: Features of `lqn_setup`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# %%
print('Example of layered model with a task paying setup and delay-off times')
model = LayeredNetwork('setup_example')
# definition of processors, tasks and entries
P1 = Processor(model, 'P1', GlobalConstants.MaxInt, SchedStrategy.INF)
T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
P2 = Processor(model, 'P2', 4, SchedStrategy.FCFS)
# Setup task: capacity, scheduling strategy, think time, setup time, delay-off time
T2 = SetupTask(model, 'F2', 6, SchedStrategy.FCFS).on(P2).set_think_time(Exp.fit_mean(8.0))
T2.setSetupTime(Exp(1.0))       # Activation time
T2.setDelayOffTime(Exp(2.0))    # Idle time before a server powers off
E2 = Entry(model, 'E2').on(T2)
# definition of activities
A1 = Activity(model, 'A1', Exp(1.0)).on(T1).bound_to(E1).synch_call(E2, 1)
A2 = Activity(model, 'A2', Exp(3.0)).on(T2).bound_to(E2).replies_to(E2)
# %%
# Define a helper function for solver selection (as in original MATLAB)
def myFunction(model):
    """Helper function to select appropriate solver for the model"""
    # This could implement logic to choose between MVA, MAM, etc.
    # For now, use MVA as default
    options = MVA.default_options()
    options.verbose = 0

    ... (3 source lines omitted; full implementation: python/examples/basic/)

lnoptions.verbose = 0
lnoptions.seed = 23000
solver = LN(model, myFunction, lnoptions)
avg_table = solver.avg_table()
print('\nLN(MVA) Results:')
print(avg_table)
```

Running this edition prints

```
Example of layered model with a task paying setup and delay-off times
```

```
LN(MVA) Results:
Node  NodeType      QLen      Util    RespT   ResidT  ArvR  Tput
P1    Processor      NaN       0.5      NaN      NaN   NaN   NaN
P2    Processor      NaN  0.041667      NaN      NaN   NaN   NaN
T1    RefTask          1       0.5      NaN        1   NaN   0.5
F2    Task           0.5  0.041667      NaN  0.33333   NaN   0.5
E1    Entry            1       0.5        2      NaN   NaN   0.5
E2    Entry          0.5  0.041667        1      NaN   NaN   0.5
A1    Activity         1       0.5        2        1   NaN   0.5
A2    Activity   0.16667  0.041667  0.33333  0.33333   NaN   0.5
```

2.7.16 `lqn_sockshop`

The Sock Shop microservice benchmark as a layered model.

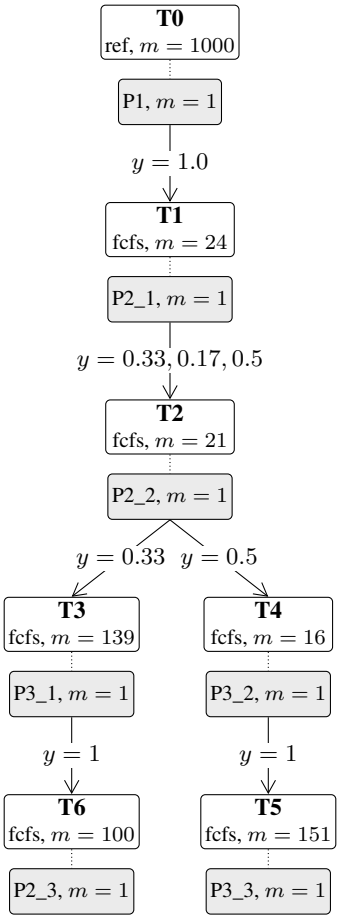

Figure 2.65: Call graph of `lqn_sockshop`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
model = LayeredNetwork('sockshop')
# Processors
P = np.empty(7, dtype=object)
P[0] = Processor(model, 'P1', 1, SchedStrategy.INF)
P[1] = Processor(model, 'P2_1', 1, SchedStrategy.PS)
P[1].setQuantum(0.1)
P[1].setReplication(2)
P[2] = Processor(model, 'P2_2', 1, SchedStrategy.PS)
P[2].setQuantum(0.1)
P[3] = Processor(model, 'P2_3', 1, SchedStrategy.PS)
P[3].setQuantum(0.1)
```

Processors	7: P1 (inf, $m = 1$), P2_1 (ps, $m = 1$), P2_2 (ps, $m = 1$), P2_3 (ps, $m = 1$), P3_1 (ps, $m = 1$), P3_2 (ps, $m = 1$), P3_3 (ps, $m = 1$)
Tasks	7: T0 (ref, $m = 1000$), T1 (fcfs, $m = 24$), T2 (fcfs, $m = 21$), T6 (fcfs, $m = 100$), T3 (fcfs, $m = 139$), T4 (fcfs, $m = 16$), T5 (fcfs, $m = 151$)
Entries	12: E0, E1, E2, E3, E4, E11, E5, E6, E7, E8, E9, E10
Activities	24, host demands AS: 5e-08, AS0: 5e-08, AS1: 5e-08, AS2: 0.0022216, AH1: 5e-08, AH2: 0.0021319, AH3: 5e-08, AH4: 0.0037561, ...
Calls	14 (14 synchronous, 0 asynchronous)
Think times	T0: 7.0

Table 2.65: Features of `lqn_sockshop`.

```
P[4] = Processor(model, 'P3_1', 1, SchedStrategy.PS)
P[4].setQuantum(0.1)
P[5] = Processor(model, 'P3_2', 1, SchedStrategy.PS)
P[5].setQuantum(0.1)
P[6] = Processor(model, 'P3_3', 1, SchedStrategy.PS)
P[6].setQuantum(0.1)
# Tasks
T = np.empty(7, dtype=object)
T[0] = Task(model, 'T0', 1000, SchedStrategy.REF).on(P[0]).set_think_time(Exp.fit_mean(7.0))
     # reference task
T[1] = Task(model, 'T1', 24, SchedStrategy.FCFS).on(P[1]).set_think_time(Immediate())    #
    edge router
T[1].setFanOut('T2', 1)
T[2] = Task(model, 'T2', 21, SchedStrategy.FCFS).on(P[2]).set_think_time(Immediate())    #
    front end
T[2].setFanOut('T3', 1)
T[2].setFanOut('T4', 1)
T[2].setFanIn('T1', 1)

    ... (77 source lines omitted; full implementation: python/examples/basic/)

model = lqn_sockshop()

# Solve using LN with MVA
avg_table_ln = LN(model).get_avg_table()
print('\nLN(MVA) Results:')
print(avg_table_ln)
```

Running this edition prints

```
LN(MVA) Results:
Node  NodeType         QLen        Util       RespT      ResidT  ArvR   Tput
P1    Processor         NaN  1.4244e-05         NaN         NaN   NaN    NaN
P2_1  Processor         NaN     0.31647         NaN         NaN   NaN    NaN
P2_2  Processor         NaN     0.55996         NaN         NaN   NaN    NaN
P2_3  Processor         NaN     0.28462         NaN         NaN   NaN    NaN
P3_1  Processor         NaN     0.43766         NaN         NaN   NaN    NaN
P3_2  Processor         NaN    0.078816         NaN         NaN   NaN    NaN
P3_3  Processor         NaN    0.078558         NaN         NaN   NaN    NaN
T0    RefTask        2.8935  1.4244e-05         NaN  1.0000e-07   NaN  142.4
T1    Task           2.8824     0.31647         NaN    0.002615   NaN  142.4
T2    Task           2.4762     0.55996         NaN   0.0079978   NaN  142.5
T6    Task          0.39567     0.28462         NaN   0.0056102   NaN   70.5
T3    Task           1.1664     0.43766         NaN    0.010929   NaN   70.5
T4    Task          0.17027    0.078816         NaN   0.0035126   NaN   24.2
T5    Task         0.085204    0.078558         NaN   0.0035178   NaN   24.2

... (28 captured-output lines omitted; rerun the source example for the full output)

AH11  Activity   4.1396e-07  2.3507e-07  1.7610e-08           0   NaN   23.5
AH12  Activity      0.27052    0.078719   0.011508   0.0019658   NaN   23.5
AS3   Activity   1.3070e-07  1.2109e-07  1.0793e-08           0   NaN   12.1
AS4   Activity     0.088246    0.042292   0.0072875   0.0018848   NaN   12.1
AS5   Activity   1.3070e-07  1.2109e-07  1.0793e-08           0   NaN   12.1
AS6   Activity     0.082021    0.036524   0.0067734   0.0016278   NaN   12.1
AH13  Activity   2.6270e-07  2.4221e-07  1.0846e-08  1.0846e-08   NaN   24.2
AH14  Activity     0.085204    0.078558   0.0035178   0.0035178   NaN   24.2
```

2.7.17 lqn_srvnph

Method srvnph: each entry activity graph composed into one phase-type service law, so an AND fork-join and a loop survive as a distribution rather than as routing.
No topology is drawn for this example: the captured run yielded no Network or LayeredNetwork to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
                    GlobalConstants, LayeredNetwork, LN, MVA, Processor,
                    SchedStrategy, Task, VerboseLevel)
model = LayeredNetwork('srvnphExample')
P1 = Processor(model, 'P1', 1, SchedStrategy.INF)
T1 = Task(model, 'T1', 20, SchedStrategy.REF).on(P1)
T1.setThinkTime(Exp.fitMean(1.0))
E1 = Entry(model, 'E1').on(T1)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T2 = Task(model, 'T2', 5, SchedStrategy.FCFS).on(P2)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T2)
P3 = Processor(model, 'P3', 1, SchedStrategy.PS)
T3 = Task(model, 'T3', 3, SchedStrategy.FCFS).on(P3)
E4 = Entry(model, 'E4').on(T3)
# client
Activity(model, 'A1', Exp.fitMean(0.1)).on(T1).boundTo(E1).synchCall(E2, 1).synchCall(E3, 1)
# E2: AND fork-join over two branches, one of which calls a third tier
A20 = Activity(model, 'A20', Exp.fitMean(0.2)).on(T2).boundTo(E2)
A21 = Activity(model, 'A21', Exp.fitMean(0.3)).on(T2)
A22 = Activity(model, 'A22', Exp.fitMean(0.2)).on(T2).synchCall(E4, 1)
A23 = Activity(model, 'A23', Exp.fitMean(0.1)).on(T2).repliesTo(E2)
T2.addPrecedence(ActivityPrecedence.AndFork(A20, [A21, A22]))
T2.addPrecedence(ActivityPrecedence.AndJoin([A21, A22], A23))
# E3: a geometric loop of mean 3 around a body of two activities
A30 = Activity(model, 'A30', Exp.fitMean(0.1)).on(T2).boundTo(E3)
A31 = Activity(model, 'A31', Exp.fitMean(0.2)).on(T2)

    ... (29 source lines omitted; full implementation: python/examples/basic/)


nclasses_default = sum(m.getNumberOfClasses() for m in solver_default.ensemble)
nclasses_srvnph = sum(m.getNumberOfClasses() for m in solver_srvnph.ensemble)
print('\nLayer classes: default %d, srvn.ph %d. Runtime: %.3fs vs %.3fs (%.2fx).'
      % (nclasses_default, nclasses_srvnph, t_default, t_srvnph,
         t_default / t_srvnph if t_srvnph > 0 else float('inf')))
```

Running this edition prints

```
[harvest] ModuleNotFoundError: No module named 'line_solver'
```

2.7.18 lqn_transient

The transient response of a layered model, through getTranAvg.

Processors	2: P1 (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	2: T1 (ref, $m = 5$), T2 (fcfs, $m = 5$)
Entries	2: E1, E2
Activities	2, host demands A1: 0.5, A2: 0.3333333333333333
Calls	1 (1 synchronous, 0 asynchronous)
Think times	T1: 1.0

Table 2.66: Features of lqn_transient.

The example is built and solved as follows.

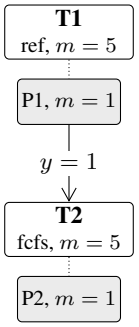

Figure 2.66: Call graph of `lqn_transient`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
                    LayeredNetwork, LN, Processor, SchedStrategy, Task,
                    VerboseLevel)
model = LayeredNetwork('lqn_transient')
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 5, SchedStrategy.REF).on(P1).setThinkTime(Exp(1.0))
T2 = Task(model, 'T2', 5, SchedStrategy.FCFS).on(P2).setThinkTime(Exp(1.0))
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
Activity(model, 'A1', Exp(2.0)).on(T1).boundTo(E1).synchCall(E2, 1)
Activity(model, 'A2', Exp(3.0)).on(T2).boundTo(E2).repliesTo(E2)
return model
"""(t, metric, station name, class name) of a transient trace entry, or None."""
if trace is None:
    return None
t = getattr(trace, 't', None)
metric = getattr(trace, 'metric', None)
if t is None and isinstance(trace, dict):
    t, metric = trace.get('t'), trace.get('metric')
if t is None or metric is None:
    return None
t = np.asarray(t, dtype=float).ravel()
metric = np.asarray(metric, dtype=float).ravel()
if metric.size == 0 or not np.any(metric > 1e-6):
    return None
handle = getattr(trace, 'handle', None)

    ... (63 source lines omitted; full implementation: python/examples/basic/)

        ax.grid(True)
        ax.legend(loc='best')
if plt is not None:
    fig.tight_layout()
    fig.savefig('lqn_transient.png', dpi=110)
    print('Wrote lqn_transient.png')
```

Running this edition prints

```
LN.getTranAvg returned transient traces for 3 layers.
/homes/gcasale/.config/matplotlib is not a writable directory
Matplotlib created a temporary cache directory at <tmp> because there was an issue with the default path
    (/homes/gcasale/.config/matplotlib); it is highly recommended to set the MPLCONFIGDIR environment
    variable to a writable directory, in particular to speed up the import of Matplotlib and to better
    support multiprocessing.
Matplotlib is building the font cache; this may take a moment.
Wrote lqn_transient.png
```

2.7.19 `lqn_twotasks`

Two tasks over separate processors, the two-layer archetype.

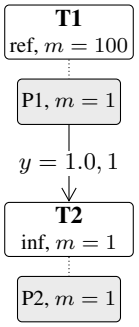

Figure 2.67: Call graph of `lqn_twotasks`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	2: T1 (ref, $m = 100$), T2 (inf, $m = 1$)
Entries	3: E1, E2, E3
Activities	5, host demands A1: 1.0, A20: 1.0, A21: 1.0, A22: 1.0, A3: 1.0
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 10.0

Table 2.67: Features of `lqn_twotasks`.

The example is built and solved as follows.

```
model = LayeredNetwork('myLayeredModel')
# Definition of processors, tasks and entries
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 100, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T2 = Task(model, 'T2', 1, SchedStrategy.INF).on(P2)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T2)
# Definition of activities
T1.set_think_time(Erlang.fit_mean_and_order(10, 1))
A1 = Activity(model, 'A1', Exp(1)).on(T1).bound_to(E1).synch_call(E2).synch_call(E3, 1)
A20 = Activity(model, 'A20', Exp(1)).on(T2).bound_to(E2)
A21 = Activity(model, 'A21', Exp(1)).on(T2)
A22 = Activity(model, 'A22', Exp(1)).on(T2).replies_to(E2)
T2.add_precedence(ActivityPrecedence.serial([A20, A21, A22]))
A3 = Activity(model, 'A3', Exp(1)).on(T2).bound_to(E3).replies_to(E3)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)
model = lqn_twotasks()

# Solve with LQNS
solver_lqns = LQNS(model, keep=True, verbose=False)
avg_table_lqns = solver_lqns.get_avg_table()
print('LQNS Results:')
print(avg_table_lqns)

# Solve with LN solver using NC as the layer solver (matches MATLAB)
# MATLAB: solver{2} = LN(model, @(l)NC(l,solveroptions), lnoptions);
solver_ln = LN(model, lambda m: NC(m, verbose=False), verbose=False)
avg_table_ln = solver_ln.get_avg_table()
print('\nNC Results:')
print(avg_table_ln)
```

Running this edition prints

```
LQNS Results:
Node  NodeType   QLen  Util  RespT  ResidT  ArvR  Tput
P1    Processor   NaN  0.25    NaN     NaN   NaN   NaN
```

```
P2    Processor   NaN     1    NaN     NaN   NaN   NaN
T1    RefTask    97.5  0.25    NaN     NaN   NaN  0.25
T2    Task       97.2     1    NaN     NaN   NaN   0.5
E1    Entry      97.5  0.25    390     NaN   NaN  0.25
E2    Entry      72.9  0.75  291.5     NaN   NaN  0.25
E3    Entry      24.3  0.25   97.2     NaN   NaN  0.25
A1    Activity   97.5  0.25    390     NaN   NaN  0.25
A20   Activity   24.3  0.25   97.2     NaN   NaN  0.25
A21   Activity   24.3  0.25   97.2     NaN   NaN  0.25
A22   Activity   24.3  0.25   97.2     NaN   NaN  0.25
A3    Activity   24.3  0.25   97.2     NaN   NaN  0.25

NC Results:

... (5 captured-output lines omitted; rerun the source example for the full output)

E1    Entry      97.5  0.25    390     NaN   NaN  0.25
E2    Entry      72.9  0.75  291.5     NaN   NaN  0.25
E3    Entry      24.3  0.25   97.2     NaN   NaN  0.25
A1    Activity   97.5  0.25    390  1.3275   NaN  0.25
A20   Activity   24.3  0.25   97.2    48.6   NaN  0.25
A21   Activity   24.3  0.25   97.2    48.6   NaN  0.25
A22   Activity   24.3  0.25   97.2    48.6   NaN  0.25
A3    Activity   24.3  0.25   97.2    48.6   NaN  0.25
```

2.7.20 `lqn_workflows`

Workflow patterns (sequence, AND-fork and join, loop) expressed as activity graphs.

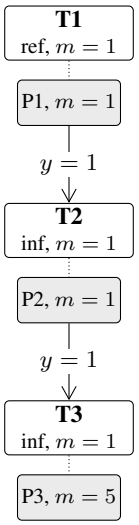

Figure 2.68: Call graph of `lqn_workflows`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	3: P1 (inf, $m = 1$), P2 (inf, $m = 1$), P3 (ps, $m = 5$)
Tasks	3: T1 (ref, $m = 1$), T2 (inf, $m = 1$), T3 (inf, $m = 1$)
Entries	3: Entry, E2, E1
Activities	14, host demands A1: 1.0, A2: 2.0, A3: 3.0, B1: 0.1, B2: 0.2, B3: 0.3, B4: 0.4, B5: 0.5, ...
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 0.0

Table 2.68: Features of `lqn_workflows`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
```

```
# %%
print('This example illustrates a layered network with a loop.')
model = LayeredNetwork('myLayeredModel')
# Layer 1 - Reference task
P1 = Processor(model, 'P1', GlobalConstants.MaxInt, SchedStrategy.INF)
T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1)
T1.set_think_time(Immediate())
E1 = Entry(model, 'Entry').on(T1)
# Layer 2 - Service task
P2 = Processor(model, 'P2', GlobalConstants.MaxInt, SchedStrategy.INF)
T2 = Task(model, 'T2', 1, SchedStrategy.INF).on(P2).set_think_time(Immediate())
E2 = Entry(model, 'E2').on(T2)
# Layer 3 - Backend task
P3 = Processor(model, 'P3', 5, SchedStrategy.PS)
T3 = Task(model, 'T3', 20, SchedStrategy.INF).on(P3)
T3.set_think_time(Exp.fit_mean(10))
E3 = Entry(model, 'E1').on(T3)  # Note: Entry named 'E1' in original
# Activities for T1 (with loop)
A1 = Activity(model, 'A1', Exp.fit_mean(1)).on(T1).bound_to(E1)
A2 = Activity(model, 'A2', Exp.fit_mean(2)).on(T1)
A3 = Activity(model, 'A3', Exp.fit_mean(3)).on(T1).synch_call(E2, 1)
# Activities for T2 (with fork-join)
B1 = Activity(model, 'B1', Exp.fit_mean(0.1)).on(T2).bound_to(E2)
B2 = Activity(model, 'B2', Exp.fit_mean(0.2)).on(T2)

    ... (27 source lines omitted; full implementation: python/examples/basic/)

    print(avg_table_lqns)
# %%
solver_ln = LN(model)
avg_table_ln = solver_ln.avg_table()
print('\nLN Results:')
print(avg_table_ln)
```

Running this edition prints

```
This example illustrates a layered network with a loop.
Warning [writeXML]: Task T3 is not a reference task, so its think time (10) is not written: the LQN XML
     schema accepts think-time on reference tasks only.

LQNS Results:
Node   NodeType        QLen       Util    RespT  ResidT  ArvR      Tput
P1     Processor        NaN    0.79575      NaN     NaN   NaN       NaN
P2     Processor        NaN    0.16711      NaN     NaN   NaN       NaN
P3     Processor        NaN   0.072414      NaN     NaN   NaN       NaN
T1     RefTask            1    0.79575      NaN     NaN   NaN  0.079575
T2     Task         0.20425    0.16711      NaN     NaN   NaN  0.079575
T3     Task        0.072396   0.072414      NaN     NaN   NaN  0.079575
Entry  Entry              1    0.79575   12.567     NaN   NaN  0.079575
E2     Entry        0.20425    0.16711   2.5667     NaN   NaN  0.079575
E1     Entry       0.072396   0.072414  0.90978     NaN   NaN  0.079575
A1     Activity    0.079575   0.079575        1     NaN   NaN  0.079575
A2     Activity     0.47745    0.47745        2     NaN   NaN   0.23873

... (30 captured-output lines omitted; rerun the source example for the full output)

B4     Activity    0.031062   0.031062      0.4     0.4   NaN  0.077655
B5     Activity    0.038827   0.038827      0.5     0.5   NaN  0.077655
B6     Activity     0.11726   0.046593     1.51     0.6   NaN  0.077655
C1     Activity   0.0078713  0.0078713      0.1     0.1   NaN  0.078713
C2     Activity   0.0047228  0.0047228      0.2    0.06   NaN  0.023614
C3     Activity   0.0070842  0.0070842      0.3    0.09   NaN  0.023614
C4     Activity    0.012594   0.012594      0.4    0.16   NaN  0.031485
C5     Activity    0.039357   0.039357      0.5     0.5   NaN  0.078713
```

2.8 Cache models

A `Cache` node holds a finite set of items under a replacement policy. Each request names an item drawn from a popularity distribution, typically Zipf; a hit is served at once, a miss must be fetched and inserted, which evicts an item. The metric of interest is the hit ratio, and everything else in the model follows

from it.

The first group of scripts fixes the workload and varies the replacement policy (LRU, FIFO, RR, CLIMB, H-LRU, Q-LRU), so the hit ratios are directly comparable. The second group builds retrieval systems, in which the cache sits in front of a backing store and the miss path is itself a queueing network, and the delayed-hit effect appears: a request for an item already being fetched waits for that fetch rather than starting another.

2.8.1 `cache_compare_replc`

Every replacement policy on one item popularity, so the hit ratios are directly comparable.

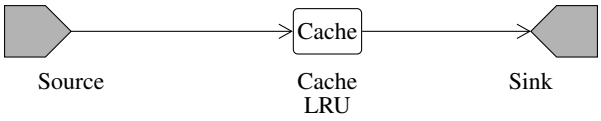

Figure 2.69: Topology of `cache_compare_replc`: an open network with a cache node.

Nodes	3: Source, Cache, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Arrivals	Source – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled
Features	cache: LRU replacement, 5 items, capacity 2,1; class switching on 2 links

Table 2.69: Features of `cache_compare_replc`.

The example is built and solved as follows.

```
n = 5
m = [2, 1]  # Two-level cache matching MATLAB
alpha = 1.0
model = Network('model')
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.RR)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(2))
p_access = Zipf(alpha, n)
cache_node.set_read(job_class, p_access)
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)
model.link(P)
return model
# Use STD verbosity to show warnings (like MATLAB behavior)
GlobalConstants.set_verbose(VerboseLevel.STD)

n = 5
m = [2, 1]  # Two-level cache matching MATLAB

    ... (97 source lines omitted; full implementation: python/examples/basic/)

        except:
            nc_hit_ratio.append(float('nan'))
    else:
        nc_hit_ratio.append(float('nan'))

    print(f'{ReplacementStrategy.to_string(repl_strat[s])}: {ctmc_hit_ratio[s]:.8f}, {
    mva_hit_ratio[s]:.8f}, {nc_hit_ratio[s]:.8f}')
```

Running this edition prints

```
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
rr: 0.71338230, 0.69673077, 0.70305657
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
fifo: 0.71338230, 0.69673077, 0.70305657
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
Warning [SolverFeatureSet]: Some features are not supported by the chosen solver (feature:
    ReplacementStrategy_LRU).
lru: 0.72430609, 0.72522055, nan
```

2.8.2 `cache_itemsize_costcap`

Per-item storage costs and a cap on the cost each list may hold, so an insertion that would breach the cap serves the request without changing the cache state.

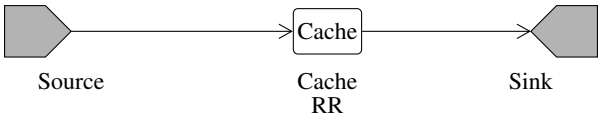

Figure 2.70: Topology of `cache_itemsize_costcap`: an open network with a cache node.

Nodes	3: Source, Cache, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Arrivals	Source – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled
Features	cache: RR replacement, 6 items, capacity 1,1; class switching on 2 links
Solvers	NC

Table 2.70: Features of `cache_itemsize_costcap`.

The example is built and solved as follows.

```
model = Network('model')
n = 6                           # number of items
m = np.array([1, 1])            # two lists, one item each
sizes = [1, 1, 1, 2, 2, 2]      # small and large items
caps = [2, 1]                   # list 2 admits small items only
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.RR)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(2))
cache_node.set_read(job_class, DiscreteSampler((1.0 / n) * np.ones(n)))
cache_node.set_item_sizes(sizes)
cache_node.set_cost_caps(caps)
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)
model.link(P)
return model, cache_node
```

```
GlobalConstants.set_verbose(VerboseLevel.STD)
model, cache_node = cache_itemsize_costcap()

solver = NC(model, 'exact')
solver.getAvgNodeTable()
solver.getAvgCacheTable()
solver.getAvgItemTable()

print("Hit Ratio:", cache_node.get_hit_ratio())
print("Mean per-list storage cost:", cache_node.get_list_cost())
```

Running this edition prints

```
NC analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Hit Ratio: [0.33333333 0.         0.         ]
Mean per-list storage cost: [1.6 1. ]
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

2.8.3 `cache_mmap_rr_env`

A marked MAP request stream into a random-replacement cache inside a random environment.

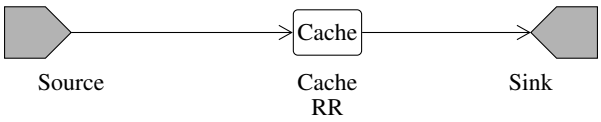

Figure 2.71: Topology of `cache_mmap_rr_env`: an open network with a cache node.

Nodes	3: Source, Cache, Sink
Classes	6: Read1 (open), Read2 (open), Hit1 (open), Miss1 (open), Hit2 (open), Miss2 (open)
Arrivals	Source – Read1: MMAP; Read2: MMAP; Hit1: Disabled; Miss1: Disabled; Hit2: Disabled; Miss2: Disabled
Features	cache: RR replacement, 4 items, capacity 2; class switching on 2 links
Solvers	LDES, CTMC, FLD

Table 2.71: Features of `cache_mmap_rr_env`.

The example is built and solved as follows.

```
          an environment modulating phase-conditional Poisson arrivals
          (D1 diagonal):
            'avg'    - fast-environment limit: replace the modulation by its
                       time-average per-class Poisson rates and solve ONE
                       rate-averaged cache;
            'dec'    - slow-environment limit: quasi-stationary
                       decomposition, solve each phase independently and
                       average per-phase hit ratios weighted by phase prob;
            'blend' - state-vector coupling: carry the cache-state
                       distribution across phase switches and average each
                       phase's sojourn-weighted distribution. For a
                       Markovian environment this recovers the exact joint
                       (cache x phase) solution, i.e. it matches CTMC.
model = Network('MMAPCache')
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.RR)
sink = Sink(model, 'Sink')
rd1 = OpenClass(model, 'Read1', 0)
rd2 = OpenClass(model, 'Read2', 0)
hit1 = OpenClass(model, 'Hit1', 0)
mis1 = OpenClass(model, 'Miss1', 0)
hit2 = OpenClass(model, 'Hit2', 0)
```

```
mis2 = OpenClass(model, 'Miss2', 0)
cache_node.set_read(rd1, p_access1)
cache_node.set_read(rd2, p_access2)
cache_node.set_hit_class(rd1, hit1)

    ... (97 source lines omitted; full implementation: python/examples/basic/)

print('\n--- Actual cache hit ratio per read class ---')
print('                  Read1      Read2')
for label, h in (('LDES (sim)  ', hit_ldes), ('CTMC (true) ', hit_ctmc),
                 ('ENV (avg)   ', hit_avg), ('ENV (dec)    ', hit_dec),
                 ('ENV (blend) ', hit_blend), ('ENV (mf/FLD)', hit_fld)):
    print('%s : %8.4f  %8.4f' % (label, h[0], h[1]))
```

Running this edition prints

```
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
      Node JobClass  QLen  Util  RespT  ResidT     ArvR     Tput
0   Source    Read1   0.0   0.0    0.0     0.0  0.00000  2.50000
1   Source    Read2   0.0   0.0    0.0     0.0  0.00000  2.50000
2    Cache    Read1   0.0   0.0    0.0     0.0  2.50000  0.00000
3    Cache    Read2   0.0   0.0    0.0     0.0  2.50000  0.00000
4    Cache     Hit1   0.0   0.0    0.0     0.0  0.00000  1.32561
5    Cache    Miss1   0.0   0.0    0.0     0.0  0.00000  1.17439
6    Cache     Hit2   0.0   0.0    0.0     0.0  0.00000  1.32298
7    Cache    Miss2   0.0   0.0    0.0     0.0  0.00000  1.17702
8     Sink     Hit1   0.0   0.0    0.0     0.0  1.32561  0.00000
9     Sink    Miss1   0.0   0.0    0.0     0.0  1.17439  0.00000
10    Sink     Hit2   0.0   0.0    0.0     0.0  1.32298  0.00000
11    Sink    Miss2   0.0   0.0    0.0     0.0  1.17702  0.00000


... (61 captured-output lines omitted; rerun the source example for the full output)

--- Actual cache hit ratio per read class ---
                  Read1      Read2
LDES (sim)    :   0.5302    0.5292
CTMC (true)   :   0.5963    0.4771
ENV (avg)     :   0.5821    0.4385
ENV (dec)     :   0.5241    0.4905
ENV (blend)   :   0.5241    0.4905
ENV (mf/FLD)  :   0.5740    0.4852
```

The rows repeat for each of the 3 solvers the script runs (LDES, CTMC, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.4 `cache_replc_climb`

The CLIMB policy, which promotes a hit by one list only.

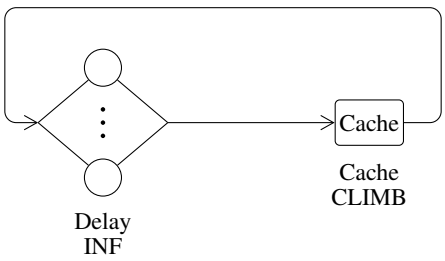

Figure 2.72: Topology of `cache_replc_climb`: a closed network over 1 queueing station with a cache node.

The example is built and solved as follows.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
```

Nodes	2: Delay, Cache
Classes	3: JobClass (closed, $N = 1$), HitClass (closed, $N = 0$), MissClass (closed, $N = 0$)
Scheduling	Delay: INF
Service	Delay – JobClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Delay; cache: CLIMB replacement, 5 items, capacity 2; class switching on 2 links
Solvers	CTMC

Table 2.72: Features of `cache_replc_climb`.

```
delay = Delay(model, 'Delay')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.CLIMB)
job_class = ClosedClass(model, 'JobClass', 1, delay, 0)
hit_class = ClosedClass(model, 'HitClass', 0, delay, 0)
miss_class = ClosedClass(model, 'MissClass', 0, delay, 0)
delay.set_service(job_class, Exp(1))
cache_node.set_read(job_class, Zipf(1.2, n))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, delay, cache_node, 1.0)
P.set(hit_class, job_class, cache_node, delay, 1.0)
P.set(miss_class, job_class, cache_node, delay, 1.0)
model.link(P)
return model, cache_node
model, cache_node = cache_replc_climb()
print(SolverCTMC(model, 'exact', keep=False).get_avg_node_table())
```

Running this edition prints

```
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: sync; type: exact, deterministic; lang: python; env: python] completed in <t>s.
                Node    JobClass  QLen  Util  RespT  ResidT         ArvR  \
0              Delay    JobClass   1.0   1.0    1.0     1.0  1.00000e+00
1              Cache    JobClass   0.0   0.0    0.0     0.0  1.00000e+00
2              Cache    HitClass   0.0   0.0    0.0     0.0  0.00000e+00
3              Cache   MissClass   0.0   0.0    0.0     0.0  0.00000e+00
4  CS_Cache_to_Delay    JobClass   0.0   0.0    0.0     0.0  2.77556e-17
5  CS_Cache_to_Delay    HitClass   0.0   0.0    0.0     0.0  5.97633e-01
6  CS_Cache_to_Delay   MissClass   0.0   0.0    0.0     0.0  4.02367e-01

      Tput
0  1.00000
1  0.00000
2  0.59763
3  0.40237
4  1.00000
5  0.00000
6  0.00000
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

2.8.5 `cache_replc_fifo`

First-in first-out replacement.
The example is built and solved as follows.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
delay = Delay(model, 'Delay')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.FIFO)
job_class = ClosedClass(model, 'JobClass', 1, delay, 0)
hit_class = ClosedClass(model, 'HitClass', 0, delay, 0)
miss_class = ClosedClass(model, 'MissClass', 0, delay, 0)
delay.set_service(job_class, Exp(1))
```

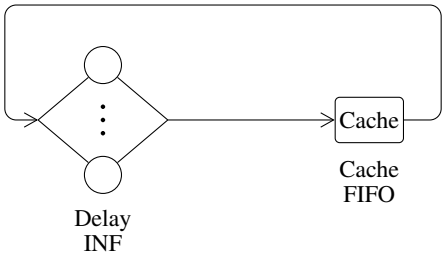

Figure 2.73: Topology of `cache_replc_fifo`: a closed network over 1 queueing station with a cache node.

Nodes	2: Delay, Cache
Classes	3: JobClass (closed, $N = 1$), HitClass (closed, $N = 0$), MissClass (closed, $N = 0$)
Scheduling	Delay: INF
Service	Delay – JobClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Delay; cache: FIFO replacement, 5 items, capacity 2; class switching on 2 links
Solvers	CTMC, SSA, MVA, FLD

Table 2.73: Features of `cache_replc_fifo`.

```
# Uniform item references
p_access = [1.0 / n] * n
cache_node.set_read(job_class, DiscreteSampler(p_access))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, delay, cache_node, 1.0)
P.set(hit_class, job_class, cache_node, delay, 1.0)
P.set(miss_class, job_class, cache_node, delay, 1.0)
model.link(P)
# Run multiple solvers
return model, cache_node
GlobalConstants.set_verbose(VerboseLevel.STD)
model, cache_node = cache_replc_fifo()
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model, keep=False))


    ... (10 source lines omitted; full implementation: python/examples/basic/)

    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_node_table[s] = solver[s].avg_node_table()
    print(avg_node_table[s])

print(f'\nHit Ratio: {cache_node.get_hit_ratio()}')
print(f'Miss Ratio: {cache_node.get_miss_ratio()}')
```

Running this edition prints

```
SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
                Node   JobClass  QLen  Util  RespT  ResidT         ArvR  Tput
0              Delay   JobClass   1.0   1.0    1.0     1.0  1.00000e+00   1.0
1              Cache   JobClass   0.0   0.0    0.0     0.0  1.00000e+00   0.0
2              Cache   HitClass   0.0   0.0    0.0     0.0  0.00000e+00   0.4
3              Cache  MissClass   0.0   0.0    0.0     0.0  0.00000e+00   0.6
4  CS_Cache_to_Delay   JobClass   0.0   0.0    0.0     0.0  2.77556e-17   1.0
5  CS_Cache_to_Delay   HitClass   0.0   0.0    0.0     0.0  4.00000e-01   0.0
6  CS_Cache_to_Delay  MissClass   0.0   0.0    0.0     0.0  6.00000e-01   0.0

SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
```

```
                Node   JobClass  QLen  Util  RespT  ResidT         ArvR  \
0              Delay   JobClass   1.0   1.0    1.0     1.0  9.97341e-01

... (32 captured-output lines omitted; rerun the source example for the full output)

Cache              HitClass      0     0      0       0            0   0.4
Cache              MissClass     0     0      0       0            0   0.6
CS_Cache_to_Delay  JobClass      0     0      0       0  5.5511e-17     1
CS_Cache_to_Delay  HitClass      0     0      0       0          0.4     0
CS_Cache_to_Delay  MissClass     0     0      0       0          0.6     0

Hit Ratio: [0.4 0.  0. ]
Miss Ratio: [0.6 0.  0. ]
```

The rows repeat for each of the 4 solvers the script runs (CTMC, SSA, MVA, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.6 `cache_replc_hlru`

H-LRU, the multi-list generalization of LRU.

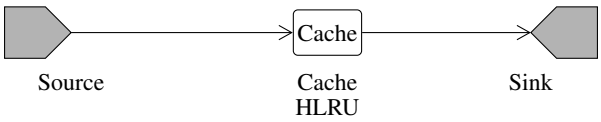

Figure 2.74: Topology of `cache_replc_hlru`: an open network with a cache node.

Nodes	3: Source, Cache, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	cache: HLRU replacement, 6 items, capacity 2,1; class switching on 2 links
Solvers	CTMC, MVA, SSA

Table 2.74: Features of `cache_replc_hlru`.

The example is built and solved as follows.

```
model = Network('model')
n = 6          # number of items
m = [2, 1]     # list capacities: list 1 holds 2 items, list 2 holds 1
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.HLRU)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
p_access = Zipf(1.2, n)  # Zipf-like item references
cache_node.set_read(job_class, p_access)
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)
model.link(P)
return model, cache_node
GlobalConstants.set_verbose(VerboseLevel.STD)
model, cache_node = cache_replc_hlru()

solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model, keep=False, cutoff=1))  # exact
```

```
    ... (5 source lines omitted; full implementation: python/examples/basic/)


avg_node_table = np.empty(len(solver), dtype=object)
for s in range(len(solver)):
    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_node_table[s] = solver[s].avg_node_table()
    print(avg_node_table[s])
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
     Node    JobClass  QLen  Util  RespT  ResidT     ArvR     Tput
0  Source   InitClass   0.0   0.0    0.0     0.0  0.00000  1.00000
1   Cache   InitClass   0.0   0.0    0.0     0.0  1.00000  0.00000
2   Cache    HitClass   0.0   0.0    0.0     0.0  0.00000  0.70085
3   Cache   MissClass   0.0   0.0    0.0     0.0  0.00000  0.29915
4    Sink    HitClass   0.0   0.0    0.0     0.0  0.70085  0.00000
5    Sink   MissClass   0.0   0.0    0.0     0.0  0.29915  0.00000

SOLVER: MVA
     Node    JobClass  QLen  Util  RespT  ResidT     ArvR     Tput

... (8 captured-output lines omitted; rerun the source example for the full output)

SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
     <t>s.
     Node    JobClass  QLen  Util  RespT  ResidT    ArvR    Tput
0  Source   InitClass   0.0   0.0    1.0     1.0  0.0000  1.0000
1   Cache   InitClass   0.0   0.0    0.0     0.0  1.0000  0.0000
2   Cache    HitClass   0.0   0.0    0.0     0.0  0.0000  0.6988
3   Cache   MissClass   0.0   0.0    0.0     0.0  0.0000  0.3012
4    Sink    HitClass   0.0   0.0    0.0     0.0  0.6988  0.0000
5    Sink   MissClass   0.0   0.0    0.0     0.0  0.3012  0.0000
```

The rows repeat for each of the 3 solvers the script runs (CTMC, MVA, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.7 `cache_replc_lru`

Least-recently-used replacement, with a closed class circulating between a Delay and the cache.

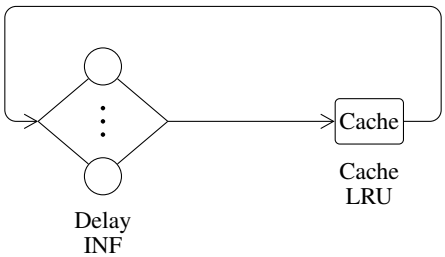

Figure 2.75: Topology of `cache_replc_lru`: a closed network over 1 queueing station with a cache node.

The example is built and solved as follows.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
delay = Delay(model, 'Delay')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.LRU)
```

Nodes	2: Delay, Cache
Classes	3: JobClass (closed, $N = 1$), HitClass (closed, $N = 0$), MissClass (closed, $N = 0$)
Scheduling	Delay: INF
Service	Delay – JobClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Delay; cache: LRU replacement, 5 items, capacity 2; class switching on 2 links
Solvers	CTMC, SSA, MVA

Table 2.75: Features of `cache_replc_lru`.

```
job_class = ClosedClass(model, 'JobClass', 1, delay, 0)
hit_class = ClosedClass(model, 'HitClass', 0, delay, 0)
miss_class = ClosedClass(model, 'MissClass', 0, delay, 0)
delay.set_service(job_class, Exp(1))
# Uniform item references
p_access = [1.0 / n] * n
cache_node.set_read(job_class, DiscreteSampler(p_access))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, delay, cache_node, 1.0)
P.set(hit_class, job_class, cache_node, delay, 1.0)
P.set(miss_class, job_class, cache_node, delay, 1.0)
model.link(P)
# Run multiple solvers
return model, cache_node
GlobalConstants.set_verbose(VerboseLevel.STD)
model, cache_node = cache_replc_lru()
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model, keep=False))


    ... (11 source lines omitted; full implementation: python/examples/basic/)

    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_node_table[s] = solver[s].avg_node_table()
    print(avg_node_table[s])

print(f'\nHit Ratio: {cache_node.get_hit_ratio()}')
print(f'Miss Ratio: {cache_node.get_miss_ratio()}')
```

Running this edition prints

```
SOLVER: CTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
                Node   JobClass  QLen  Util  RespT  ResidT         ArvR  Tput
0              Delay   JobClass   1.0   1.0    1.0     1.0  1.00000e+00   1.0
1              Cache   JobClass   0.0   0.0    0.0     0.0  1.00000e+00   0.0
2              Cache   HitClass   0.0   0.0    0.0     0.0  0.00000e+00   0.4
3              Cache  MissClass   0.0   0.0    0.0     0.0  0.00000e+00   0.6
4  CS_Cache_to_Delay   JobClass   0.0   0.0    0.0     0.0  2.77556e-17   1.0
5  CS_Cache_to_Delay   HitClass   0.0   0.0    0.0     0.0  4.00000e-01   0.0
6  CS_Cache_to_Delay  MissClass   0.0   0.0    0.0     0.0  6.00000e-01   0.0

SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
                Node   JobClass  QLen  Util  RespT  ResidT         ArvR  \
0              Delay   JobClass   1.0   1.0    1.0     1.0  9.97341e-01

... (21 captured-output lines omitted; rerun the source example for the full output)

2              Cache   HitClass   0.0   0.0    0.0     0.0  0.00000e+00   0.4
3              Cache  MissClass   0.0   0.0    0.0     0.0  0.00000e+00   0.6
4  CS_Cache_to_Delay   JobClass   0.0   0.0    0.0     0.0  5.55112e-17   1.0
5  CS_Cache_to_Delay   HitClass   0.0   0.0    0.0     0.0  4.00000e-01   0.0
6  CS_Cache_to_Delay  MissClass   0.0   0.0    0.0     0.0  6.00000e-01   0.0

Hit Ratio: [0.4 0.  0. ]
Miss Ratio: [0.6 0.  0. ]
```

The rows repeat for each of the 3 solvers the script runs (CTMC, SSA, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.8 `cache_replc_qlru`

Q-LRU, which promotes a hit with probability q.

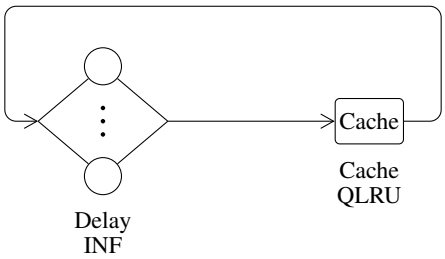

Figure 2.76: Topology of `cache_replc_qlru`: a closed network over 1 queueing station with a cache node.

Nodes	2: Delay, Cache
Classes	3: JobClass (closed, $N = 1$), HitClass (closed, $N = 0$), MissClass (closed, $N = 0$)
Scheduling	Delay: INF
Service	Delay – JobClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Delay; cache: QLRU replacement, 5 items, capacity 2; class switching on 2 links
Solvers	CTMC

Table 2.76: Features of `cache_replc_qlru`.

The example is built and solved as follows.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
delay = Delay(model, 'Delay')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.QLRU)
cache_node.set_admission_prob(0.5)  # admit a missed item with probability q=0.5
job_class = ClosedClass(model, 'JobClass', 1, delay, 0)
hit_class = ClosedClass(model, 'HitClass', 0, delay, 0)
miss_class = ClosedClass(model, 'MissClass', 0, delay, 0)
delay.set_service(job_class, Exp(1))
cache_node.set_read(job_class, Zipf(1.2, n))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, delay, cache_node, 1.0)
P.set(hit_class, job_class, cache_node, delay, 1.0)
P.set(miss_class, job_class, cache_node, delay, 1.0)
model.link(P)
return model, cache_node
model, cache_node = cache_replc_qlru()
print(SolverCTMC(model, 'exact', keep=False).get_avg_node_table())
```

Running this edition prints

```
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: sync; type: exact, deterministic; lang: python; env: python] completed in <t>s.
               Node   JobClass  QLen  Util  RespT  ResidT         ArvR  \
0             Delay   JobClass   1.0   1.0    1.0     1.0  1.00000e+00
1             Cache   JobClass   0.0   0.0    0.0     0.0  1.00000e+00
```

```
2              Cache   HitClass   0.0   0.0    0.0     0.0  0.00000e+00
3              Cache  MissClass   0.0   0.0    0.0     0.0  0.00000e+00
4  CS_Cache_to_Delay   JobClass   0.0   0.0    0.0     0.0  5.55112e-17
5  CS_Cache_to_Delay   HitClass   0.0   0.0    0.0     0.0  5.75143e-01
6  CS_Cache_to_Delay  MissClass   0.0   0.0    0.0     0.0  4.24857e-01

      Tput
0  1.00000
1  0.00000
2  0.57514
3  0.42486
4  1.00000
5  0.00000
6  0.00000
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

2.8.9 `cache_replc_routing`

Cache-aware routing: hits and misses leave the cache on different arcs, under RAND and round-robin routing.

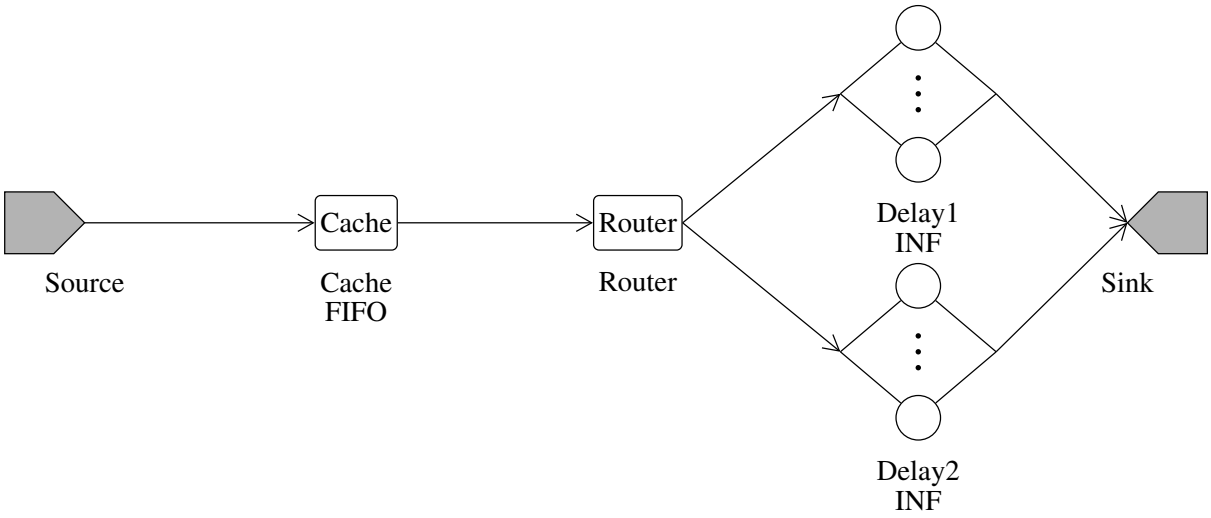

Figure 2.77: Topology of `cache_replc_routing`: an open network over 2 queueing stations with a cache node and a router.

Nodes	6: Source, Delay × 2, Cache, Router, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Scheduling	Delay1: INF; Delay2: INF
Arrivals	Source – InitClass: Exp(2)
Service	Delay1 – HitClass: Exp(10); MissClass: Exp(1) Delay2 – HitClass: Exp(20); MissClass: Exp(2)
Features	cache: FIFO replacement, 5 items, capacity 2; class switching on 6 links
Solvers	CTMC, SSA, MVA, NC, FLD

Table 2.77: Features of `cache_replc_routing`.

The example is built and solved as follows.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.FIFO)
router_node = Router(model, 'Router')
```

```
delay1 = Delay(model, 'Delay1')
delay2 = Delay(model, 'Delay2')
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(2))
# Different service times for hits and misses
delay1.set_service(hit_class, Exp(10))
delay1.set_service(miss_class, Exp(1))
delay2.set_service(hit_class, Exp(20))
delay2.set_service(miss_class, Exp(2))
# Uniform item references
p_access = [1.0 / n] * n
cache_node.set_read(job_class, DiscreteSampler(p_access))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
# Set up routing
model.add_link(source, cache_node)
model.add_link(cache_node, router_node)

    ... (35 source lines omitted; full implementation: python/examples/basic/)

    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_node_table[s] = solver[s].avg_node_table()
    print(avg_node_table[s])

print(f'\nHit Ratio: {cache_node.get_hit_ratio()}')
print(f'Miss Ratio: {cache_node.get_miss_ratio()}')
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
Warning [solver_ctmc]: CTMC: 40 vanishing state(s) have no immediate outgoing arc; the vanishing predicate
      and the immediate-arc tagging disagree, so those rows keep their timed arcs.
  Stochcomp: 1040 immediate states eliminated, 960 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
      Node    JobClass      QLen      Util  RespT  ResidT      ArvR      Tput
0   Source   InitClass  0.00000  0.00000   0.00    0.00  0.00000  1.98079
1    Cache   InitClass  0.00000  0.00000   0.00    0.00  1.98079  0.00000
2    Cache    HitClass  0.00000  0.00000   0.00    0.00  0.00000  0.79232
3    Cache   MissClass  0.00000  0.00000   0.00    0.00  0.00000  1.18847
4   Router    HitClass  0.00000  0.00000   0.00    0.00  0.79232  0.79232
5   Router   MissClass  0.00000  0.00000   0.00    0.00  1.18847  1.18847
6   Delay1    HitClass  0.03896  0.03896   0.10    0.02  0.39616  0.38955

... (64 captured-output lines omitted; rerun the source example for the full output)

Delay1  MissClass     0     0      0       0   0.6     0
Delay2  HitClass      0     0      0       0   0.4     0
Delay2  MissClass     0     0      0       0   0.6     0
Sink    HitClass      0     0      0       0   0.8     0
Sink    MissClass     0     0      0       0   1.2     0

Hit Ratio: [0.4 0.  0. ]
Miss Ratio: [0.6 0.  0. ]
```

The rows repeat for each of the 5 solvers the script runs (CTMC, SSA, MVA, NC, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.10 `cache_replc_rr`

Random replacement.
The example is built and solved as follows.

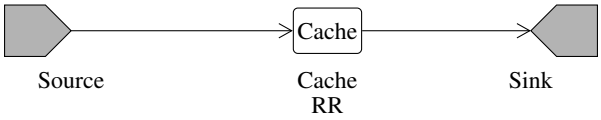

Figure 2.78: Topology of `cache_replc_rr`: an open network with a cache node.

Nodes	3: Source, Cache, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Arrivals	Source – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled
Features	cache: RR replacement, 5 items, capacity 2; class switching on 2 links
Solvers	CTMC, SSA, MVA, NC, FLD

Table 2.78: Features of `cache_replc_rr`.

```
model = Network('model')
n = 5  # number of items
m = 2  # cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, m, ReplacementStrategy.RR)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(2))
# Zipf-like item references
p_access = Zipf(1.4, n)
cache_node.set_read(job_class, p_access)
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)
model.link(P)
# Run multiple solvers
return model, cache_node
GlobalConstants.set_verbose(VerboseLevel.STD)
model, cache_node = cache_replc_rr()
solver = np.array([], dtype=object)
solver = np.append(solver, CTMC(model, keep=False, cutoff=1))

    ... (15 source lines omitted; full implementation: python/examples/basic/)

    print(f'\nSOLVER: {solver[s].get_name()}')
    avg_node_table[s] = solver[s].avg_node_table()
    print(avg_node_table[s])

print(f'\nHit Ratio: {cache_node.get_hit_ratio()}')
print(f'Miss Ratio: {cache_node.get_miss_ratio()}')
```

Running this edition prints

```
SOLVER: CTMC
CTMC solver using state space cutoff = 1 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
     Node    JobClass  QLen  Util  RespT  ResidT     ArvR     Tput
0  Source   InitClass   0.0   0.0    0.0     0.0  0.00000  2.00000
1   Cache   InitClass   0.0   0.0    0.0     0.0  2.00000  0.00000
2   Cache    HitClass   0.0   0.0    0.0     0.0  0.00000  1.18079
3   Cache   MissClass   0.0   0.0    0.0     0.0  0.00000  0.81921
4    Sink    HitClass   0.0   0.0    0.0     0.0  1.18079  0.00000
5    Sink   MissClass   0.0   0.0    0.0     0.0  0.81921  0.00000
```

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.

... (31 captured-output lines omitted; rerun the source example for the full output)

Cache   InitClass     0     0      0       0        2        0
Cache   HitClass      0     0      0       0        0   1.1246
Cache   MissClass     0     0      0       0        0  0.87542
Sink    HitClass      0     0      0       0   1.1246        0
Sink    MissClass     0     0      0       0  0.87542        0

Hit Ratio: [0.56229204 0.         0.        ]
Miss Ratio: [0.43770796 0.         0.        ]
```

The rows repeat for each of the 5 solvers the script runs (CTMC, SSA, MVA, NC, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.11 `cache_rmf_transient`

The transient hit ratio from the refined mean field approximation.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# Cache parameters
n = 10        # number of items
m = [3, 2]    # list capacities (2-list cache)
alpha = 0.8   # Zipf exponent
# Zipf popularity distribution
p = np.array([(i + 1) ** (-alpha) for i in range(n)])
p = p / p.sum()
print(f'Cache parameters: n={n}, m=[{", ".join(str(x) for x in m)}], Zipf({alpha:.1f})')
# Build DDPP model
model = CacheRMF(p, m)
# Steady-state analysis with 1/N correction
pi, V, (V_full, W) = model.meanFieldExpansionSteadyState(order=1)
pi_refined = pi + V / n
print('\nSteady-state results (refined mean field):')
total_hit = 0.0
for k in range(1, len(m) + 1):
    hr = model.hit_rate(pi_refined, k)
    total_hit += hr
    print(f'  Hit rate (list {k}): {hr:.6f}')
miss_rate = model.hit_rate(pi_refined, 0)
print(f'  Miss rate:          {miss_rate:.6f}')
print(f'  Total hit prob:     {total_hit:.6f}')
print(f'  Total miss prob:    {miss_rate:.6f}')
# Transient analysis
T, X, Vt, Wt = model.meanFieldExpansionTransient(time=50.0, n_points=200, order=1)
print(f'\nTransient hit rates (refined, N={n}):')
time_indices = [0, 20, 50, 100, 199]  # t=0, ~5, ~12.5, ~25, 50
for idx in time_indices:
    xt = X[idx, :] + Vt[idx, :] / n
    hr = sum(model.hit_rate(xt, k) for k in range(1, len(m) + 1))
    print(f'  t={T[idx]:7.3f}: hit_rate={hr:.6f}')
cache_rmf_transient()
```

Running this edition prints

```
Cache parameters: n=10, m=[3, 2], Zipf(0.8)

Steady-state results (refined mean field):
  Hit rate (list 1): 0.313605
  Hit rate (list 2): 0.300298
  Miss rate:         0.386096
  Total hit prob:    0.613904
  Total miss prob:   0.386096
```

```
Transient hit rates (refined, N=10):
  t=  0.000: hit_rate=0.728003
  t=  5.025: hit_rate=0.645750
  t= 12.563: hit_rate=0.624411
  t= 25.126: hit_rate=0.616758
  t= 50.000: hit_rate=0.614587
```

2.8.12 `cachenet_tandem`

A tandem of two caches over one item set, where a miss at the first is a read of the same item at the second. Item identity is carried by a dedicated job class per item, so no class switching happens on the arc between the caches.

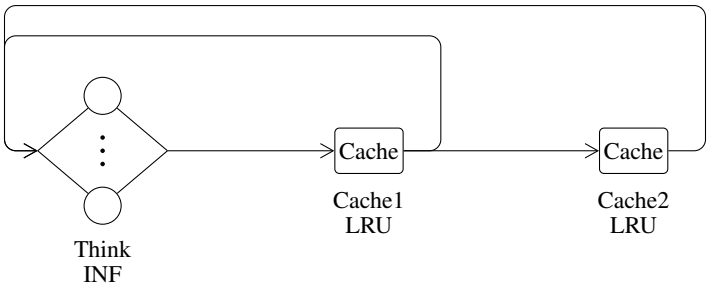

Figure 2.79: Topology of `cachenet_tandem`: a closed network over 1 queueing station with a cache node.

Nodes	3: Delay, Cache × 2
Classes	15: Read1 (closed, $N = 1$), Hit1_1 (closed, $N = 0$), Hit2_1 (closed, $N = 0$), Miss_1 (closed, $N = 0$), Read2 (closed, $N = 1$), Hit1_2 (closed, $N = 0$), Hit2_2 (closed, $N = 0$), Miss_2 (closed, $N = 0$), Read3 (closed, $N = 1$), Hit1_3 (closed, $N = 0$), Hit2_3 (closed, $N = 0$), Miss_3 (closed, $N = 0$), Cache2_item1 (closed, $N = 0$), Cache2_item2 (closed, $N = 0$), Cache2_item3 (closed, $N = 0$)
Scheduling	Think: INF
Service	Think – Read1: Exp(0.5); Hit1_1: Disabled; Hit2_1: Disabled; Miss_1: Disabled; Read2: Exp(0.3); Hit1_2: Disabled; Hit2_2: Disabled; Miss_2: Disabled; Read3: Exp(0.2); Hit1_3: Disabled; Hit2_3: Disabled; Miss_3: Disabled; Cache2_item1: Disabled; Cache2_item2: Disabled; Cache2_item3: Disabled
Features	drop rule {'Hit1_1': 'waitingQueue', 'Hit2_1': 'waitingQueue', 'Miss_1': 'waitingQueue', 'Hit1_2': 'waitingQueue', 'Hit2_2': 'waitingQueue', 'Miss_2': 'waitingQueue', 'Hit1_3': 'waitingQueue', 'Hit2_3': 'waitingQueue', 'Miss_3': 'waitingQueue', 'Cache2_item1': 'waitingQueue', 'Cache2_item2': 'waitingQueue', 'Cache2_item3': 'waitingQueue'} at Think; cache: LRU replacement, 3 items, capacity 1; cache: LRU replacement, 3 items, capacity 1; class switching on 3 links
Solvers	CTMC, SSA, LDES

Table 2.79: Features of `cachenet_tandem`.

The example is built and solved as follows.

```
"""Build the two-level cache network and return (model, cache1, cache2)."""
n = 3                           # items, shared by both caches
p_access = [0.5, 0.3, 0.2]      # item popularity
lambda_ = 1.0                   # request rate scale
cap1, cap2 = 1, 1               # items held by Cache1, Cache2
model = Network('CacheTandem')
think = Delay(model, 'Think')
cache1 = Cache(model, 'Cache1', n, cap1, ReplacementStrategy.LRU)
cache2 = Cache(model, 'Cache2', n, cap2, ReplacementStrategy.LRU)
# One dedicated class per item for each role.
```

```
read, hit1, hit2, miss = [], [], [], []
for f in range(n):
    read.append(ClosedClass(model, f'Read{f + 1}', 1, think, 0))
    hit1.append(ClosedClass(model, f'Hit1_{f + 1}', 0, think, 0))
    hit2.append(ClosedClass(model, f'Hit2_{f + 1}', 0, think, 0))
    miss.append(ClosedClass(model, f'Miss_{f + 1}', 0, think, 0))
    think.set_service(read[f], Exp(lambda_ * p_access[f]))
cache1.set_item_read_classes(read, hit1)        # read[f] reads item f at Cache1
cache1.set_miss_cache(read[0], cache2, hit2)    # Cache1 miss of item f = Cache2 read of item
    f
cache2.set_item_miss_class(read[0], miss)       # Cache2 miss leaves the network
P = model.init_routing_matrix()
for f in range(n):
    P.set(read[f], read[f], think, cache1, 1.0)
    P.set(hit1[f], read[f], cache1, think, 1.0)    # hit at Cache1
    P.set(hit2[f], read[f], cache2, think, 1.0)    # hit at Cache2
    P.set(miss[f], read[f], cache2, think, 1.0)    # miss at both

    ... (9 source lines omitted; full implementation: python/examples/basic/)


model.reset()
print(SSA(model, samples=50000, seed=1).getAvgCacheTable())

model.reset()
print(LDES(model, samples=200000, seed=1).getAvgCacheTable())
```

Running this edition prints

```
Warning [solver_ctmc]: The generator has more than one closed communicating class (351 of 2277 states
    cannot reach the one containing the initial state), so its stationary distribution is not unique.
    Results are those of the class the initial state selects, and depend on it: an order-preserving
    discipline under routing that admits no overtaking freezes the relative service order at time zero,
    so declaring the classes in a different order gives a different, equally valid answer. Product-form
    results (MVA, NC) normalize over the whole space instead and will not agree.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Node    JobClass      List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate  ArvR  ResidT  ListCost
Cache1  Read1            0        1      3      0.5               0       0.5        0               0
          0     0     NaN       NaN
Cache1  Read2            0        1      3      0.3               0       0.7        0               0
          0     0     NaN       NaN
Cache1  Read3            0        1      3      0.2               0       0.8        0               0
          0     0     NaN       NaN
Cache2  Cache2_item1     0        1      3  0.33333               0   0.66667        0               0
          0     0     NaN       NaN
Cache2  Cache2_item2     0        1      3  0.33333               0   0.66667        0               0
          0     0     NaN       NaN
Cache2  Cache2_item3     0        1      3  0.33333               0   0.66667        0               0
          0     0     NaN       NaN
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
Node    JobClass      List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate  ArvR  ResidT  ListCost
Cache1  Read1            0        1      3  0.49686               0   0.50314        0               0
          0     0     NaN       NaN
Cache1  Read2            0        1      3  0.29673               0   0.70327        0               0
          0     0     NaN       NaN
Cache1  Read3            0        1      3  0.20091               0   0.79909        0               0
          0     0     NaN       NaN
Cache2  Cache2_item1     0        1      3        0               0         0        0               0
          0     0     NaN       NaN
Cache2  Cache2_item2     0        1      3        0               0         0        0               0
          0     0     NaN       NaN
Cache2  Cache2_item3     0        1      3        0               0         0        0               0
          0     0     NaN       NaN
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 1
Node    JobClass      List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate  ArvR  ResidT  ListCost
Cache1  Read1            0        1      3  0.49878               0   0.50122        0               0
          0     0     NaN       NaN
Cache1  Read2            0        1      3  0.29924               0   0.70076        0               0
          0     0     NaN       NaN
Cache1  Read3            0        1      3  0.20015               0   0.79985        0               0
```

```
          0     0     NaN       NaN
Cache2  Cache2_item1     0       1      3       0              0       1       0              0
          0     0     NaN       NaN
Cache2  Cache2_item2     0       1      3       0              0       1       0              0
          0     0     NaN       NaN
Cache2  Cache2_item3     0       1      3       0              0       1       0              0
          0     0     NaN       NaN
```

The rows repeat for each of the 3 solvers the script runs (CTMC, SSA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.13 `retrieval_chain`

A chain of caches, each backing the one before it.

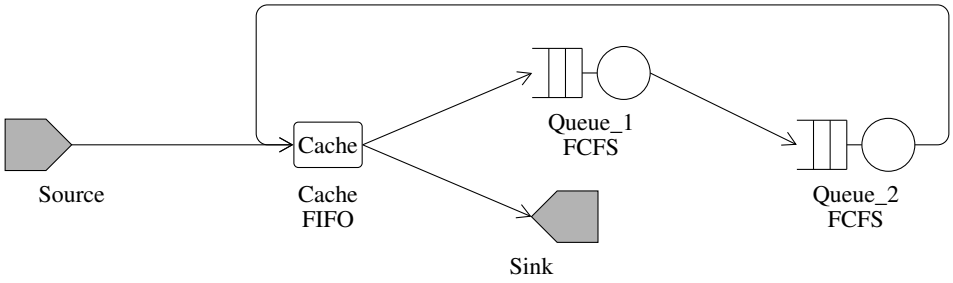

Figure 2.80: Topology of `retrieval_chain`: an open network over 2 queueing stations with a cache node.

Nodes	5: Source, Queue × 2, Cache, Sink
Classes	6: InitClass (open), HitClass (open), MissClass (open), InitClass_retrievalClass_1 (open), InitClass_retrievalClass_2 (open), InitClass_retrievalClass_3 (open)
Scheduling	Queue_1: FCFS; Queue_2: FCFS
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Disabled; InitClass_retrievalClass_2: Disabled; InitClass_retrievalClass_3: Disabled
Service	Queue_1 – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(2); InitClass_retrievalClass_2: Exp(2); InitClass_retrievalClass_3: Exp(2) Queue_2 – InitClass: Exp(3); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(3); InitClass_retrievalClass_2: Exp(3); InitClass_retrievalClass_3: Exp(3)
Features	cache: FIFO replacement, 3 items, capacity 1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue_1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue_2; class switching on 4 links
Solvers	SSA, LDES, MVA, NC

Table 2.80: Features of `retrieval_chain`.

The example is built and solved as follows.

```
access_prob = [0.6, 0.3, 0.1]          # per-item access probabilities (3 items)
n_queues = 2
model = Network('Chain Model')
n = len(access_prob)                   # number of items
capacity = [1]                         # per-level cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, capacity, ReplacementStrategy.FIFO)
queues = [Queue(model, f'Queue_{i + 1}', SchedStrategy.FCFS) for i in range(n_queues)]
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
```

```
# Read class service at each retrieval queue = default per-item fetch service.
queues[0].set_service(job_class, Exp(2.0))
queues[1].set_service(job_class, Exp(3.0))
cache_node.set_read(job_class, DiscreteSampler(access_prob))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
# No service_rates and no routing_matrices: both inherited from the read class.
cache_node.set_retrieval_system(job_class, miss_class, queues)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
# Retrieval chain drawn once for the read class: Cache -> Queue_1 -> Queue_2 -> Cache
P.set(job_class, job_class, cache_node, queues[0], 1.0)
P.set(job_class, job_class, queues[0], queues[1], 1.0)

    ... (17 source lines omitted; full implementation: python/examples/basic/)

print('SOLVER: NC')
print(NC(model).get_avg_cache_table().to_string(index=False))

# Item-level cache occupancy
print(MVA(model).get_avg_item_table())
print(NC(model).get_avg_item_table())
```

Running this edition prints

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Warning [solver_ssa_analyzer_serial]: Retrieval-system expected latency is not currently implemented;
    reporting NaN.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.43536         0.14651   0.41813  0.43536         0.14651
    0.41813  0.56464     NaN       NaN
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 1000000 --seed 1
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.43654         0.14457   0.41889  0.43654         0.14457
    0.41889  0.56346  0.83753       NaN
SOLVER: MVA
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.40688         0.16346   0.42966  0.40688         0.16346
    0.42966  0.59312  0.87138       NaN
SOLVER: NC
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.43729         0.14466   0.41805  0.43729         0.14466
    0.41805  0.56271     NaN       NaN
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN  0.47287   NaN             NaN                 NaN
Cache     2     1        1   NaN   0.3522   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.17492   NaN             NaN                 NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN  0.54346   NaN             NaN                 NaN
Cache     2     1        1   NaN   0.3278   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.12873   NaN             NaN                 NaN
```

The rows repeat for each of the 4 solvers the script runs (SSA, LDES, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.14 `retrieval_default`

The same system with a PS backing store and the default retrieval parameters.
The example is built and solved as follows.

```
access_prob = [0.6, 0.3, 0.1]          # per-item access probabilities (3 items)
model = Network('DelayedHits')
```

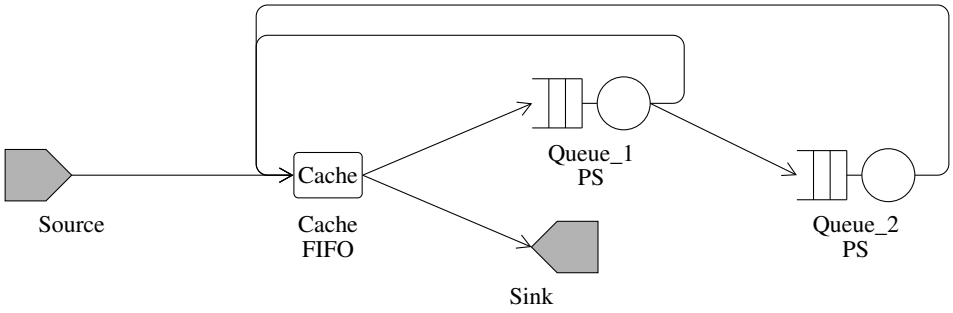

Figure 2.81: Topology of `retrieval_default`: an open network over 2 queueing stations with a cache node.

Nodes	5: Source, Queue × 2, Cache, Sink
Classes	6: InitClass (open), HitClass (open), MissClass (open), InitClass_retrievalClass_1 (open), InitClass_retrievalClass_2 (open), InitClass_retrievalClass_3 (open)
Scheduling	Queue_1: PS; Queue_2: PS
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Disabled; InitClass_retrievalClass_2: Disabled; InitClass_retrievalClass_3: Disabled
Service	Queue_1 – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(5); InitClass_retrievalClass_2: Exp(2); InitClass_retrievalClass_3: Exp(2) Queue_2 – InitClass: Exp(3); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(3); InitClass_retrievalClass_2: Exp(3); InitClass_retrievalClass_3: Exp(3)
Features	cache: FIFO replacement, 3 items, capacity 1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue_1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue_2; class switching on 4 links
Solvers	SSA, LDES, MVA, NC

Table 2.81: Features of `retrieval_default`.

```
n = len(access_prob)                    # number of items
capacity = [1]                          # per-level cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, capacity, ReplacementStrategy.FIFO)
# PS retrieval stations: per-item (class-dependent) service rates are admissible
# in the analytical retrieval algorithm (FCFS/SIRO would require identical rates).
queue1 = Queue(model, 'Queue_1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue_2', SchedStrategy.PS)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
# Read class service at each retrieval queue = default per-item fetch service.
queue1.set_service(job_class, Exp(2.0))
queue2.set_service(job_class, Exp(3.0))
cache_node.set_read(job_class, DiscreteSampler(access_prob))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
# No service_rates and no routing_matrices: both inherited from the read class.
cache_node.set_retrieval_system(job_class, miss_class, [queue1, queue2])
# Item-level overrides for item 0: skip Queue_2 and fetch faster at Queue_1.
cache_node.set_item_routing_prob(job_class, 0, queue1, queue2, 0.0)  # delete default edge
cache_node.set_item_routing_prob(job_class, 0, queue1, cache_node, 1.0)  # exit after Queue_1

    ... (23 source lines omitted; full implementation: python/examples/basic/)

print('SOLVER: NC')
print(NC(model).get_avg_cache_table().to_string(index=False))

# Item-level cache occupancy
print(MVA(model).get_avg_item_table())
print(NC(model).get_avg_item_table())
```

Running this edition prints

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Warning [solver_ssa_analyzer_serial]: Retrieval-system expected latency is not currently implemented;
    reporting NaN.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.46965        0.075575   0.45477  0.46965        0.075575
    0.45477  0.53035     NaN       NaN
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 1000000 --seed 1
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.46309        0.078601   0.45831  0.46309        0.078601
    0.45831  0.53691  0.57536       NaN
SOLVER: MVA
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.42401        0.084524   0.49147  0.42401        0.084524
    0.49147  0.57599  0.54262       NaN
SOLVER: NC
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.46383        0.078456   0.45771  0.46383        0.078456
    0.45771  0.53617     NaN       NaN
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN  0.51803   NaN             NaN                 NaN
Cache     2     1        1   NaN  0.32496   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.15701   NaN             NaN                 NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN  0.61737   NaN             NaN                 NaN
Cache     2     1        1   NaN  0.27574   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.10689   NaN             NaN                 NaN
```

The rows repeat for each of the 4 solvers the script runs (SSA, LDES, MVA, NC), which is the point of

the example: the same model read by methods of different cost and different exactness.

2.8.15 `retrieval_ps`

The PS variant, with the delayed-hit behaviour made explicit.

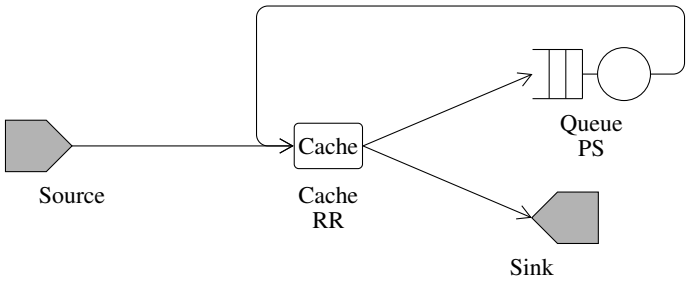

Figure 2.82: Topology of `retrieval_ps`: an open network over 1 queueing station with a cache node.

Nodes	4: Source, Queue, Cache, Sink
Classes	10: InitClass (open), HitClass (open), MissClass (open), InitClass_retrievalClass_1 (open), InitClass_retrievalClass_2 (open), InitClass_retrievalClass_3 (open), InitClass_retrievalClass_4 (open), InitClass_retrievalClass_5 (open), InitClass_retrievalClass_6 (open), InitClass_retrievalClass_7 (open)
Scheduling	Queue: PS
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Disabled; InitClass_retrievalClass_2: Disabled; InitClass_retrievalClass_3: Disabled; InitClass_retrievalClass_4: Disabled; InitClass_retrievalClass_5: Disabled; InitClass_retrievalClass_6: Disabled; InitClass_retrievalClass_7: Disabled
Service	Queue – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(1); InitClass_retrievalClass_2: Exp(1); InitClass_retrievalClass_3: Exp(1); InitClass_retrievalClass_4: Exp(1); InitClass_retrievalClass_5: Exp(1); InitClass_retrievalClass_6: Exp(1); InitClass_retrievalClass_7: Exp(1)
Features	cache: RR replacement, 7 items, capacity 6; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue; class switching on 3 links
Solvers	SSA, LDES, MVA, NC

Table 2.82: Features of `retrieval_ps`.

The example is built and solved as follows.

```
access_prob = list(np.array([49, 49, 49, 49, 7, 1, 1]) / 205)  # per-item arrival
    probabilities (7 items)
model = Network('PS Model')
n = len(access_prob)                # number of items
capacity = [6]                      # per-level cache capacity (total m = 6)
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, capacity, ReplacementStrategy.RR)
queue = Queue(model, 'Queue', SchedStrategy.PS)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
# Read class service at the retrieval queue = default per-item fetch service.
queue.set_service(job_class, Exp(1.0))
cache_node.set_read(job_class, DiscreteSampler(access_prob))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
# No service_rates and no routing_matrices: both inherited from the read class.
cache_node.set_retrieval_system(job_class, miss_class, queue)
```

```
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
# Retrieval topology drawn once for the read class: cache <-> queue.
P.set(job_class, job_class, cache_node, queue, 1.0)
P.set(job_class, job_class, queue, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)

    ... (14 source lines omitted; full implementation: python/examples/basic/)

print('SOLVER: NC')
print(NC(model).get_avg_cache_table().to_string(index=False))

# Item-level cache occupancy
print(MVA(model).get_avg_item_table())
print(NC(model).get_avg_item_table())
```

Running this edition prints

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Warning [solver_ssa_analyzer_serial]: Retrieval-system expected latency is not currently implemented;
    reporting NaN.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR  ResidT  ListCost
Cache  InitClass     0        6      7  0.98297       0.0016451   0.01538  0.98297       0.0016451
    0.01538  0.017026     NaN       NaN
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 1000000 --seed 1
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR   ResidT  ListCost
Cache  InitClass     0        6      7  0.98326       0.0020547   0.01468  0.98326       0.0020547
    0.01468  0.016735  0.98923       NaN
SOLVER: MVA
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR  ResidT  ListCost
Cache  InitClass     0        6      7  0.97573       0.0034469  0.020826  0.97573       0.0034469
    0.020826  0.024273  1.0155       NaN
SOLVER: NC
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR  ResidT  ListCost
Cache  InitClass     0        6      7  0.98272       0.0021597  0.015118  0.98272       0.0021597
    0.015118  0.017278     NaN       NaN

... (9 captured-output lines omitted; rerun the source example for the full output)

Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        6   NaN   0.9888   NaN             NaN                 NaN
Cache     2     1        6   NaN   0.9888   NaN             NaN                 NaN
Cache     3     1        6   NaN   0.9888   NaN             NaN                 NaN
Cache     4     1        6   NaN   0.9888   NaN             NaN                 NaN
Cache     5     1        6   NaN  0.93459   NaN             NaN                 NaN
Cache     6     1        6   NaN   0.5551   NaN             NaN                 NaN
Cache     7     1        6   NaN   0.5551   NaN             NaN                 NaN
```

The rows repeat for each of the 4 solvers the script runs (SSA, LDES, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.16 `retrieval_routing`

Retrieval with routing that separates the hit and miss paths.
The example is built and solved as follows.

```
access_prob = [0.6, 0.3, 0.1]          # per-item access probabilities (3 items)
model = Network('Probabilistic Routing')
n = len(access_prob)                   # number of items
capacity = [2]                         # per-level cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, capacity, ReplacementStrategy.FIFO)
is_queue = Queue(model, 'IS Queue', SchedStrategy.INF)
```

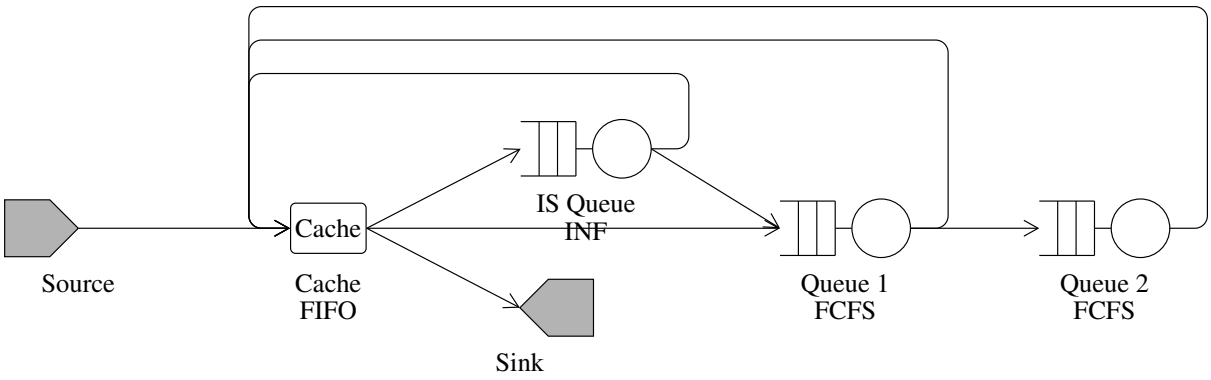

Figure 2.83: Topology of `retrieval_routing`: an open network over 3 queueing stations with a cache node.

Nodes	6: Source, Queue × 3, Cache, Sink
Classes	6: InitClass (open), HitClass (open), MissClass (open), InitClass_retrievalClass_1 (open), InitClass_retrievalClass_2 (open), InitClass_retrievalClass_3 (open)
Scheduling	IS Queue: INF; Queue 1: FCFS; Queue 2: FCFS
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Disabled; InitClass_retrievalClass_2: Disabled; InitClass_retrievalClass_3: Disabled
Service	IS Queue – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(2); InitClass_retrievalClass_2: Exp(2); InitClass_retrievalClass_3: Exp(2) Queue 1 – InitClass: Exp(3); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(3); InitClass_retrievalClass_2: Exp(3); InitClass_retrievalClass_3: Exp(3) Queue 2 – InitClass: Exp(3); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(3); InitClass_retrievalClass_2: Exp(3); InitClass_retrievalClass_3: Exp(3)
Features	cache: FIFO replacement, 3 items, capacity 2; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at IS Queue; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue 1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue 2; class switching on 5 links
Solvers	SSA, LDES, MVA, NC

Table 2.83: Features of `retrieval_routing`.

```
queue1 = Queue(model, 'Queue 1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue 2', SchedStrategy.FCFS)
queues = [is_queue, queue1, queue2]
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
# Read class service per queue = default per-item fetch service (uniform over items).
is_queue.set_service(job_class, Exp(2.0))
queue1.set_service(job_class, Exp(3.0))
queue2.set_service(job_class, Exp(3.0))
cache_node.set_read(job_class, DiscreteSampler(access_prob))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
cache_node.set_retrieval_system(job_class, miss_class, queues)
# Per-item routing over [IS(0), Queue1(1), Queue2(2), Cache(3)], applied via the
# per-item routing methods. Index n_queues (last row/col, 0-based) is the cache;
# row = from, col = to.

    ... (40 source lines omitted; full implementation: python/examples/basic/)

print('SOLVER: NC')
print(NC(model).get_avg_cache_table().to_string(index=False))

# Item-level cache occupancy
print(MVA(model).get_avg_item_table())
print(NC(model).get_avg_item_table())
```

Running this edition prints

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Warning [solver_ssa_analyzer_serial]: Retrieval-system expected latency is not currently implemented;
     reporting NaN.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        2      3  0.78014        0.037736   0.18212  0.78014        0.037736
     0.18212  0.21986     NaN       NaN
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 1000000 --seed 1
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        2      3  0.78385        0.038516   0.17764  0.78385        0.038516
     0.17764  0.21615  0.61413       NaN
SOLVER: MVA
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        2      3  0.74517        0.050821   0.20401  0.74517        0.050821
     0.20401  0.25483  0.62901       NaN
SOLVER: NC
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
     >s.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        2      3  0.78352        0.038664   0.17781  0.78352        0.038664
     0.17781  0.21648     NaN       NaN
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        2   NaN  0.80366   NaN             NaN                 NaN
Cache     2     1        2   NaN  0.71671   NaN             NaN                 NaN
Cache     3     1        2   NaN  0.47963   NaN             NaN                 NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
     >s.
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        2   NaN  0.85864   NaN             NaN                 NaN
Cache     2     1        2   NaN  0.77102   NaN             NaN                 NaN
Cache     3     1        2   NaN  0.37034   NaN             NaN                 NaN
```

The rows repeat for each of the 4 solvers the script runs (SSA, LDES, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.8.17 `retrieval_simple`

A retrieval system: a cache in front of an infinite-server backing store, hits served at once and misses fetched.

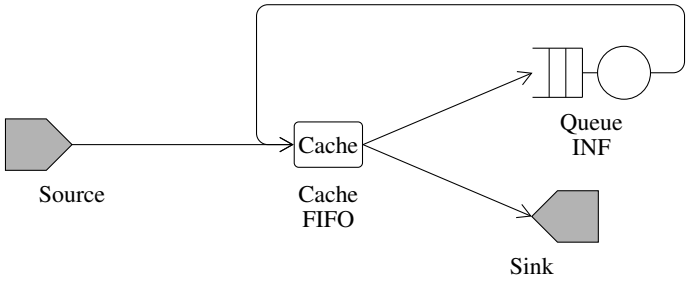

Figure 2.84: Topology of `retrieval_simple`: an open network over 1 queueing station with a cache node.

Nodes	4: Source, Queue, Cache, Sink
Classes	6: InitClass (open), HitClass (open), MissClass (open), InitClass_retrievalClass_1 (open), InitClass_retrievalClass_2 (open), InitClass_retrievalClass_3 (open)
Scheduling	Queue: INF
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Disabled; InitClass_retrievalClass_2: Disabled; InitClass_retrievalClass_3: Disabled
Service	Queue – InitClass: Exp(2); HitClass: Disabled; MissClass: Disabled; InitClass_retrievalClass_1: Exp(2); InitClass_retrievalClass_2: Exp(2); InitClass_retrievalClass_3: Exp(2)
Features	cache: FIFO replacement, 3 items, capacity 1; drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Queue; class switching on 3 links
Solvers	SSA, LDES, MVA, NC

Table 2.84: Features of `retrieval_simple`.

The example is built and solved as follows.

```
access_prob = [0.6, 0.3, 0.1]      # per-item access probabilities (3 items)
model = Network('Simple Model')
n = len(access_prob)               # number of items
capacity = [1]                     # per-level cache capacity
source = Source(model, 'Source')
cache_node = Cache(model, 'Cache', n, capacity, ReplacementStrategy.FIFO)
queue = Queue(model, 'Queue', SchedStrategy.INF)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'InitClass', 0)
hit_class = OpenClass(model, 'HitClass', 0)
miss_class = OpenClass(model, 'MissClass', 0)
source.set_arrival(job_class, Exp(1))
# Read class service at the retrieval queue = default per-item fetch service.
queue.set_service(job_class, Exp(2.0))
cache_node.set_read(job_class, DiscreteSampler(access_prob))
cache_node.set_hit_class(job_class, hit_class)
cache_node.set_miss_class(job_class, miss_class)
# No service_rates and no routing_matrices: both inherited from the read class.
cache_node.set_retrieval_system(job_class, miss_class, queue)
P = model.init_routing_matrix()
P.set(job_class, job_class, source, cache_node, 1.0)
# Retrieval topology drawn once for the read class: cache <-> queue.
P.set(job_class, job_class, cache_node, queue, 1.0)
P.set(job_class, job_class, queue, cache_node, 1.0)
P.set(hit_class, hit_class, cache_node, sink, 1.0)
P.set(miss_class, miss_class, cache_node, sink, 1.0)

    ... (14 source lines omitted; full implementation: python/examples/basic/)
```

```
print('SOLVER: NC')
print(NC(model).get_avg_cache_table().to_string(index=False))

# Item-level cache occupancy
print(MVA(model).get_avg_item_table())
print(NC(model).get_avg_item_table())
```

Running this edition prints

```
SOLVER: SSA
SSA analysis [method: serial; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Warning [solver_ssa_analyzer_serial]: Retrieval-system expected latency is not currently implemented;
     reporting NaN.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.45153        0.092363   0.45611  0.45153        0.092363
     0.45611  0.54847     NaN       NaN
SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 1000000 --seed 1
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.44672        0.091368   0.46191  0.44672        0.091368
     0.46191  0.55328  0.4999       NaN
SOLVER: MVA
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.41334         0.10048   0.48618  0.41334         0.10048
     0.48618  0.58666     0.5       NaN
SOLVER: NC
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
     >s.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.44605        0.091157    0.4628  0.44605        0.091157
     0.4628  0.55395     NaN       NaN
Node   Item  List  ListCap  Size    Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN   0.487   NaN             NaN                 NaN
Cache     2     1        1   NaN  0.3492   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.1638   NaN             NaN                 NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <t
     >s.
Node   Item  List  ListCap  Size     Prob  Cost  DelayedHitQLen  DelayedHitQLenFull
Cache     1     1        1   NaN  0.56447   NaN             NaN                 NaN
Cache     2     1        1   NaN  0.31905   NaN             NaN                 NaN
Cache     3     1        1   NaN  0.11648   NaN             NaN                 NaN
```

The rows repeat for each of the 4 solvers the script runs (SSA, LDES, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.9 Cluster models

A cluster is a dispatcher in front of a pool of servers. It is not a new node type: the shorthand builds a Router and a set of stations, and the interest is in the dispatching policy and in how the response time scales with the pool size.

2.9.1 `cl_basic`

A dispatcher in front of a pool of identical servers, built with the cluster shorthand.
The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

lambda_rates = np.array([0.4])
D = np.full((4, 1), 1.0)  # mean service time = 1 at each server

model = Network.cluster_ps(lambda_rates, D, dispatching=RoutingStrategy.RAND)
print(MVA(model).get_avg_table())
```

Running this edition prints

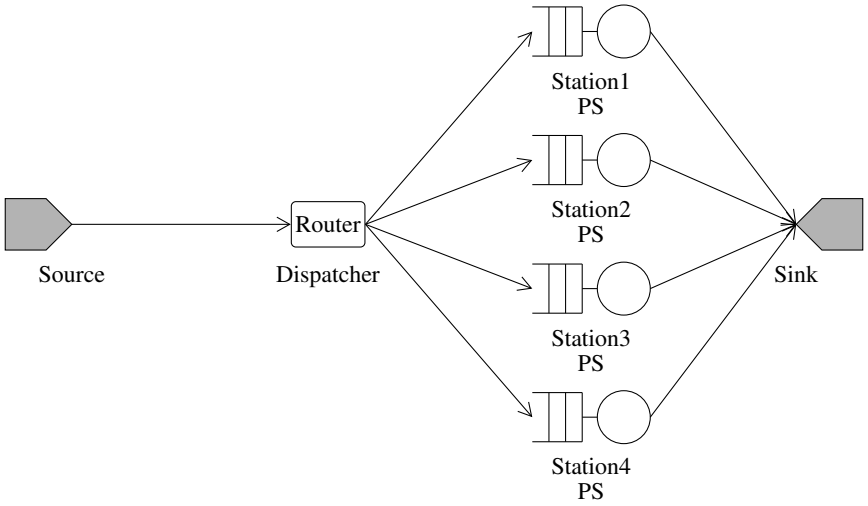

Figure 2.85: Topology of `cl_basic`: an open network over 4 queueing stations with a router.

Nodes	7: Source, Queue × 4, Router, Sink
Classes	1: Class1 (open)
Scheduling	Station1: PS; Station2: PS; Station3: PS; Station4: PS
Arrivals	Source – Class1: Exp(0.4)
Service	Station1 – Class1: Exp(1) Station2 – Class1: Exp(1) Station3 – Class1: Exp(1) Station4 – Class1: Exp(1)
Features	class switching on 7 links
Solvers	MVA

Table 2.85: Features of `cl_basic`.

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   JobClass     QLen  Util   RespT   ResidT  ArvR  Tput
Source    Class1          0     0       0        0     0   0.4
Station1  Class1    0.10977   0.1  1.0977  0.27442   0.1   0.1
Station2  Class1    0.10977   0.1  1.0977  0.27442   0.1   0.1
Station3  Class1    0.10977   0.1  1.0977  0.27442   0.1   0.1
Station4  Class1    0.10977   0.1  1.0977  0.27442   0.1   0.1
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.9.2 `cl_closed`

The closed version, with a fixed population of requests.

Nodes	5: Queue × 3, Delay, Router
Classes	1: Class1 (closed, $N = 8$)
Scheduling	Think: INF; Station1: PS; Station2: PS; Station3: PS
Service	Think – Class1: Exp(1) Station1 – Class1: Exp(1) Station2 – Class1: Exp(1) Station3 – Class1: Exp(1)
Features	class switching on 5 links
Solvers	MVA

Table 2.86: Features of `cl_closed`.

The example is built and solved as follows.

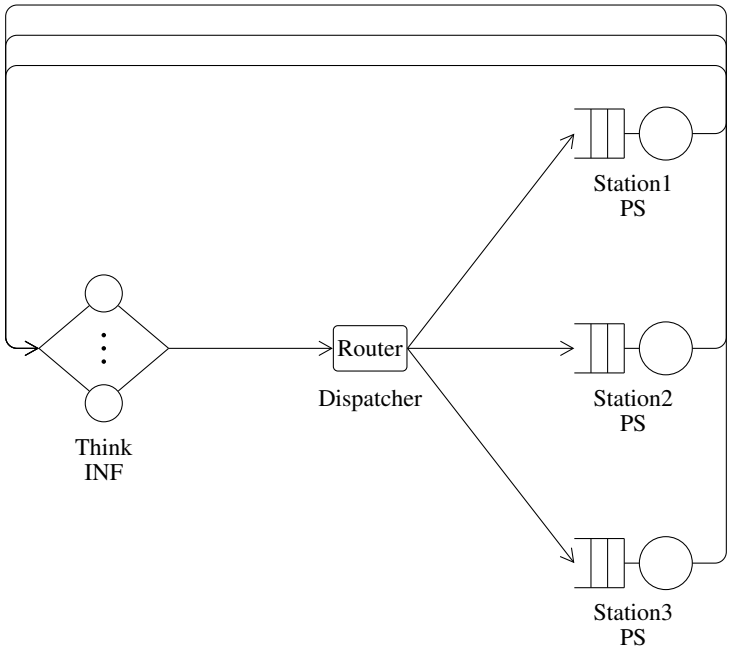

Figure 2.86: Topology of `cl_closed`: a closed network over 4 queueing stations with a router.

```
                    SchedStrategy, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.STD)

N = np.array([8])            # population
Z = np.array([1.0])           # think time
D = np.full((3, 1), 1.0)     # mean service time at each server
strategies = [SchedStrategy.PS] * 3

model = Network.cluster_closed(N, Z, D, strategies, dispatching=RoutingStrategy.RAND)
print(MVA(model).get_avg_table())
```

Running this edition prints

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass    QLen     Util   RespT   ResidT     ArvR     Tput
Think     Class1    2.1998   2.1998       1        1   2.1998   2.1998
Station1  Class1    1.9334  0.73326  2.6367  0.87891  0.73326  0.73326
Station2  Class1    1.9334  0.73326  2.6367  0.87891  0.73326  0.73326
Station3  Class1    1.9334  0.73326  2.6367  0.87891  0.73326  0.73326
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.9.3 `cl_compare`

RAND against round-robin dispatching on one cluster.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	SSA

Table 2.87: Features of `cl_compare`.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

cluster = (Cluster().set_num_stations(4).set_arrival_rate(1.0).set_service_rate(0.4)
        .set_scheduling(SchedStrategy.PS))

def solve(model):
    return SSA(model, seed=23000, samples=2000).get_avg_table()

results = cluster.compare_dispatching(
    solve,
    [RoutingStrategy.RAND, RoutingStrategy.RROBIN])
for policy, table in results.items():
    print(f"\n=== Dispatching: {policy} ===")
    print(table)
```

Running this edition prints

```
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.

=== Dispatching: RoutingStrategy.RAND ===
Station   JobClass     QLen     Util    RespT   ResidT  ArvR     Tput
Source    Class1          0        0        0        0     0        1
Station1  Class1     3.0285  0.75571   10.019   2.5047  0.25  0.30228
Station2  Class1    0.97566  0.50451   4.8347   1.2087  0.25   0.2018
Station3  Class1      1.521  0.61834   6.1495   1.5374  0.25  0.24734
Station4  Class1     1.0996  0.59497   4.6202   1.1551  0.25  0.23799

=== Dispatching: RoutingStrategy.RROBIN ===
Station   JobClass     QLen     Util    RespT   ResidT  ArvR     Tput
Source    Class1          0        0        0        0     0        1
Station1  Class1    0.98681  0.61328   4.0227   1.0057  0.25  0.24531
Station2  Class1     1.4066  0.70065    5.019   1.2547  0.25  0.28026
Station3  Class1     1.2067  0.66059   4.5666   1.1416  0.25  0.26424
Station4  Class1     1.3752  0.73211   4.6961    1.174  0.25  0.29284
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `SSA`.

2.9.4 `cl_multiclass`

Several request classes through one cluster.

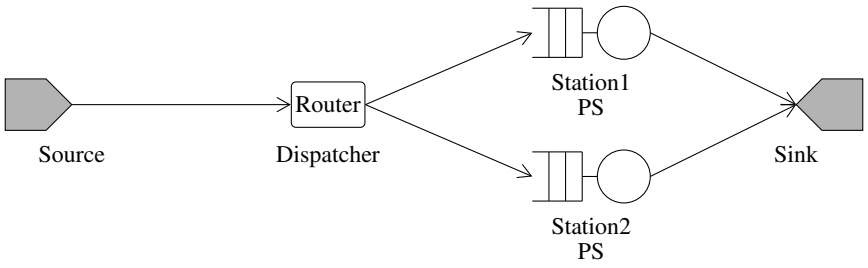

Figure 2.87: Topology of `cl_multiclass`: an open network over 2 queueing stations with a router.

The example is built and solved as follows.

```
GlobalConstants.set_verbose(VerboseLevel.STD)

lambda_rates = np.array([0.3, 0.2])            # two classes
D = np.array([[1.0, 0.5], [1.0, 0.5]])        # M=2 servers, R=2 classes

model = Network.cluster_ps(lambda_rates, D, dispatching=RoutingStrategy.RAND)
print(MVA(model).get_avg_table())
```

Nodes	5: Source, Queue × 2, Router, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Station1: PS; Station2: PS
Arrivals	Source – Class1: Exp(0.3); Class2: Exp(0.2)
Service	Station1 – Class1: Exp(1); Class2: Exp(2) Station2 – Class1: Exp(1); Class2: Exp(2)
Features	class switching on 5 links
Solvers	MVA

Table 2.88: Features of `cl_multiclass`.

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   JobClass      QLen  Util    RespT   ResidT  ArvR  Tput
Source    Class1          0     0        0        0     0   0.3
Source    Class2          0     0        0        0     0   0.2
Station1  Class1     0.18711  0.15   1.2474  0.62368  0.15  0.15
Station1  Class2    0.062368  0.05  0.62368  0.31184   0.1   0.1
Station2  Class1     0.18711  0.15   1.2474  0.62368  0.15  0.15
Station2  Class2    0.062368  0.05  0.62368  0.31184   0.1   0.1
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.9.5 `cl_scheduling`

The same cluster under FCFS and under PS.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	SSA

Table 2.89: Features of `cl_scheduling`.

The example is built and solved as follows.

```
                     Cluster, SSA, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.STD)

cluster = Cluster().set_num_stations(3).set_arrival_rate(0.9).set_service_rate(0.5)
cluster.set_dispatching(RoutingStrategy.RAND)
cluster.set_service_scv(4.0)

def solve(model):
    return SSA(model, seed=23000, samples=20000).get_avg_table()

disciplines = [SchedStrategy.FCFS, SchedStrategy.PS]
results = cluster.compare_scheduling(solve, disciplines)

for discipline, table in results.items():
    print(f"\n=== Scheduling: {getattr(discipline, 'name', discipline)} ===")
    print(table)
```

Running this edition prints

```
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
     <t>s.
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
     <t>s.

=== Scheduling: FCFS ===
Station   JobClass    QLen     Util   RespT  ResidT  ArvR     Tput
```

```
Source    Class1         0        0       0       0     0      0.9
Station1  Class1    2.4717  0.58059  8.3616  2.7872   0.3   0.2956
Station2  Class1    3.7389  0.63624  12.425  4.1416   0.3  0.30092
Station3  Class1    2.0762  0.58284  7.1433  2.3811   0.3  0.29065

=== Scheduling: PS ===
Station   JobClass    QLen     Util   RespT  ResidT  ArvR     Tput
Source    Class1         0        0       0       0     0      0.9
Station1  Class1    1.6389  0.62872  5.3583  1.7861   0.3  0.30586
Station2  Class1    1.8167  0.62477  5.9541  1.9847   0.3  0.30512
Station3  Class1    1.3415  0.58477   4.598  1.5327   0.3  0.29175
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `SSA`.

2.9.6 `cl_stations`

Clusters whose members are declared station by station.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	MVA

Table 2.90: Features of `cl_stations`.

The example is built and solved as follows.

```
                      Cluster, MVA, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.STD)

cluster = Cluster().set_num_stations(1).set_arrival_rate(1.6).set_service_rate(1.0)
cluster.set_scheduling(SchedStrategy.PS).set_dispatching(RoutingStrategy.RAND)

results = cluster.sweep_num_stations([2, 3, 4, 6], MVA)

for count in sorted(results):
    table = results[count]
    # the Source row carries no service, so it is excluded from the summary
    rows = table[table['Station'] != 'Source']
    print(f"\n=== M = {count}  (mean RespT {rows['RespT'].mean():.4f}, "
          f"max Util {rows['Util'].max():.4f}) ===")
    print(table)
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

=== M = 2  (mean RespT 5.0000, max Util 0.8000) ===
Station   JobClass  QLen  Util  RespT  ResidT  ArvR  Tput
Source    Class1       0     0      0       0     0   1.6
Station1  Class1       4   0.8      5     2.5   0.8   0.8
Station2  Class1       4   0.8      5     2.5   0.8   0.8

=== M = 3  (mean RespT 2.1429, max Util 0.5333) ===
Station   JobClass    QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1         0        0       0        0        0      1.6
Station1  Class1    1.1429  0.53333  2.1429  0.71429  0.53333  0.53333
Station2  Class1    1.1429  0.53333  2.1429  0.71429  0.53333  0.53333

... (11 captured-output lines omitted; rerun the source example for the full output)
```

```
Station   JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1          0        0       0        0        0      1.6
Station1  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
Station2  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
Station3  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
Station4  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
Station5  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
Station6  Class1    0.36351  0.26667  1.3632  0.22719  0.26667  0.26667
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.9.7 `cl_sweep`

A sweep over the cluster size, and the response time curve it traces.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	MVA

Table 2.91: Features of `cl_sweep`.

The example is built and solved as follows.

```
                     Cluster, MVA, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.STD)

cluster = (Cluster().set_num_stations(2).set_arrival_rate(0.1).set_service_rate(1.0)
        .set_scheduling(SchedStrategy.PS)
        .set_dispatching(RoutingStrategy.RAND))

rates = [0.2, 0.5, 0.9, 1.5]
results = cluster.sweep_arrival_rate(rates, MVA)
for lam, table in results.items():
    print(f"\n=== lambda = {lam} ===")
    print(table)
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

=== lambda = 0.2 ===
Station   JobClass     QLen  Util   RespT   ResidT  ArvR  Tput
Source    Class1          0     0       0        0     0   0.2
Station1  Class1    0.10977   0.1  1.0977  0.54885   0.1   0.1
Station2  Class1    0.10977   0.1  1.0977  0.54885   0.1   0.1

=== lambda = 0.5 ===
Station   JobClass     QLen  Util   RespT   ResidT  ArvR  Tput
Source    Class1          0     0       0        0     0   0.5
Station1  Class1    0.33314  0.25  1.3326  0.66629  0.25  0.25
Station2  Class1    0.33314  0.25  1.3326  0.66629  0.25  0.25

... (4 captured-output lines omitted; rerun the source example for the full output)

Station1  Class1    0.81818  0.45  1.8182  0.90909  0.45  0.45
Station2  Class1    0.81818  0.45  1.8182  0.90909  0.45  0.45

=== lambda = 1.5 ===
Station   JobClass  QLen  Util  RespT  ResidT  ArvR  Tput
Source    Class1       0     0      0       0     0   1.5
Station1  Class1       3  0.75      4       2  0.75  0.75
```

```
Station2  Class1       3  0.75      4       2  0.75  0.75
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

2.9.8 `dispatch_closed`

The closed version.

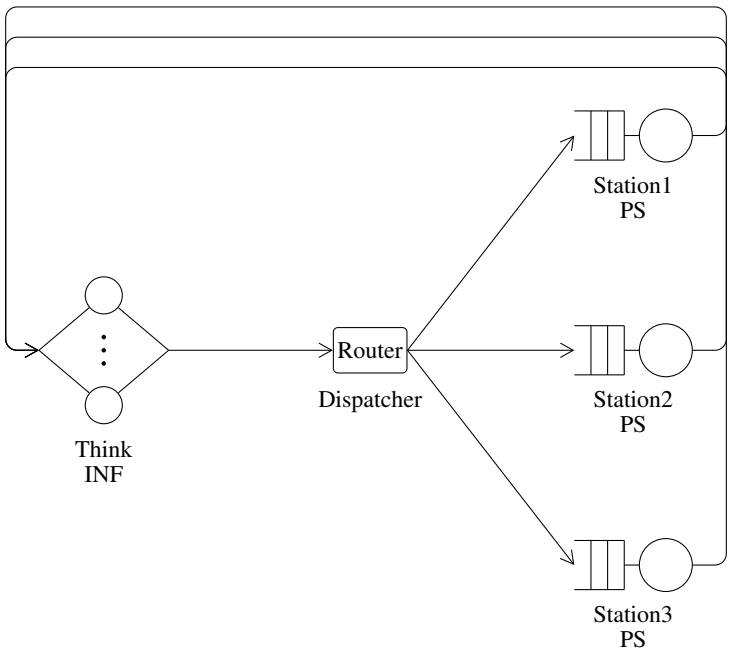

Figure 2.88: Topology of `dispatch_closed`: a closed network over 4 queueing stations with a router.

Nodes	5: Queue × 3, Delay, Router
Classes	1: Class1 (closed, $N = 10$)
Scheduling	Think: INF; Station1: PS; Station2: PS; Station3: PS
Service	Think – Class1: Exp(1) Station1 – Class1: Exp(1) Station2 – Class1: Exp(1) Station3 – Class1: Exp(1)
Features	class switching on 5 links
Solvers	MVA, SSA

Table 2.92: Features of `dispatch_closed`.

The example is built and solved as follows.

```
3 PS server stations, single closed class with N = 10 jobs, think time
Z = 1.0, mu = 1.0, RAND dispatching. Solved exactly via MVA, then
re-solved by stochastic simulation via SSA for cross-validation.
GlobalConstants.set_verbose(VerboseLevel.STD)
cluster = (Cluster()
        .set_num_stations(3)
        .set_service_rate(1.0)
        .set_scheduling(SchedStrategy.PS)
        .set_closed(10, 1.0))
# Bound rather than built inline: the parity rows serialize the example's own
# Network, and an unbound `cluster.build()` leaves them nothing to export.
model = cluster.build()
model2 = cluster.build()
```

```
print(MVA(model).get_avg_table())
print(SSA(model2, seed=23000, samples=20000).get_avg_table())
```

Running this edition prints

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
Think     Class1      2.36     2.36       1       1     2.36     2.36
Station1  Class1    2.5467  0.78666  3.2373  1.0791  0.78666  0.78666
Station2  Class1    2.5467  0.78666  3.2373  1.0791  0.78666  0.78666
Station3  Class1    2.5467  0.78666  3.2373  1.0791  0.78666  0.78666
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
Station   JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
Think     Class1    2.3383   2.3383       1       1   2.3977   2.3383
Station1  Class1    2.5789   0.7926  3.2537  1.0846  0.77944   0.7926
Station2  Class1    2.6247  0.82577  3.1785  1.0595  0.77944  0.82577
Station3  Class1    2.4581  0.77929  3.1542  1.0514  0.77944  0.77929
```

The rows repeat for each of the 2 solvers the script runs (MVA, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.9.9 `dispatch_mixed`

The mixed version, open and closed classes through the same dispatcher.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	MVA, LDES

Table 2.93: Features of `dispatch_mixed`.

The example is built and solved as follows.

```
2 PS server stations shared by one open class (lambda = 0.5) and one closed
class (N = 3 jobs, think time Z = 1.0), mu = 2.0 for the open class and
mu = 1.5 for the closed one, RAND dispatching. Solved by MVA and re-solved
by simulation via LDES for cross-validation.
GlobalConstants.set_verbose(VerboseLevel.STD)
cluster = (Cluster()
        .set_num_stations(2)
        .set_mixed(0.5, 3, 1.0)
        .set_service_rates([[2.0, 1.5], [2.0, 1.5]])
        .set_scheduling(SchedStrategy.PS))
print(MVA(cluster.build()).get_avg_table())
print(LDES(cluster.build(), seed=23000).get_avg_table())
```

Running this edition prints

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1          0        0       0        0        0      0.5
Think     Class2     1.4095   1.4095       1        1   1.4095   1.4095
Station1  Class1    0.25646    0.125  1.0258  0.51292     0.25     0.25
Station1  Class2    0.79523  0.46985  1.1284  0.56418  0.70477  0.70477
Station2  Class1    0.25646    0.125  1.0258  0.51292     0.25     0.25
Station2  Class2    0.79523  0.46985  1.1284  0.56418  0.70477  0.70477
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station   JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1          0        0       0        0        0      0.5
Think     Class2     1.4137   1.4055  1.0052   1.0052   1.4285   1.4055
Station1  Class1    0.25256  0.12377  1.0222  0.51112  0.25126  0.24726
Station1  Class2    0.79556   0.4696   1.131  0.56551  0.71574  0.70412
Station2  Class1    0.25386  0.12307  1.0144  0.50718  0.25381  0.25003
Station2  Class2    0.79073  0.46636  1.1264   0.5632  0.71276  0.70132
```

The rows repeat for each of the 2 solvers the script runs (MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.9.10 `dispatch_open`

The dispatcher idiom written out in full, open workload.

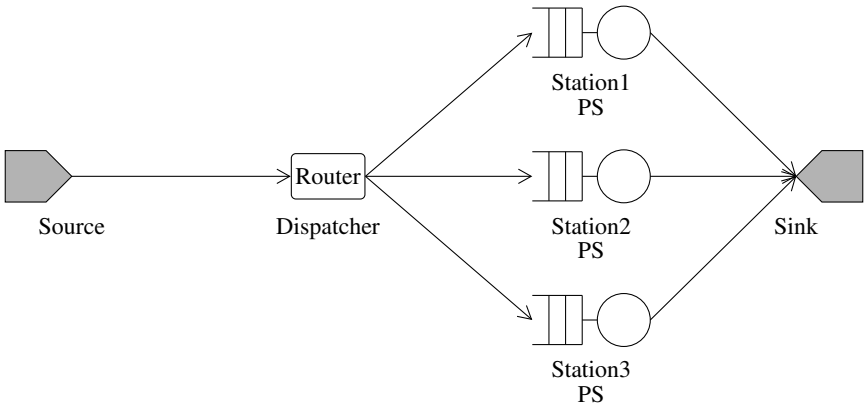

Figure 2.89: Topology of `dispatch_open`: an open network over 3 queueing stations with a router.

Nodes	6: Source, Queue × 3, Router, Sink
Classes	1: Class1 (open)
Scheduling	Station1: PS; Station2: PS; Station3: PS
Arrivals	Source – Class1: Exp(0.4)
Service	Station1 – Class1: Exp(1) Station2 – Class1: Exp(1) Station3 – Class1: Exp(1)
Features	class switching on 6 links
Solvers	MVA, SSA

Table 2.94: Features of `dispatch_open`.

The example is built and solved as follows.

```
3 PS server stations, single open class, lambda = 0.4, mu = 1.0, RAND
dispatching. Solved exactly via MVA, then re-solved by stochastic
simulation via SSA for cross-validation.
GlobalConstants.set_verbose(VerboseLevel.STD)
cluster = (Cluster()
        .set_num_stations(3)
        .set_arrival_rate(0.4)
        .set_service_rate(1.0)
        .set_scheduling(SchedStrategy.PS))
model = cluster.build()
print(MVA(model).get_avg_table())
print(SSA(model, seed=23000, samples=20000).get_avg_table())
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station   JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1          0        0       0        0        0      0.4
Station1  Class1    0.15263  0.13333  1.1448  0.38158  0.13333  0.13333
Station2  Class1    0.15263  0.13333  1.1448  0.38158  0.13333  0.13333
Station3  Class1    0.15263  0.13333  1.1448  0.38158  0.13333  0.13333
SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
     <t>s.
Station   JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source    Class1          0        0       0        0        0      0.4
Station1  Class1    0.15055  0.13402  1.1234  0.37447  0.13333  0.13402
```

```
Station2  Class1    0.15023  0.13001  1.1556   0.3852  0.13333  0.13001
Station3  Class1    0.15926  0.14042  1.1342  0.37805  0.13333  0.14042
```

The rows repeat for each of the 2 solvers the script runs (MVA, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

2.10 Stochastic Petri nets

A stochastic Petri net is built from `Place` and `Transition` nodes. A transition fires when its input places hold enough tokens, consuming and producing tokens according to the arc weights; timed transitions fire after a random delay, immediate transitions fire at once, and an inhibitor arc disables a transition while its place is marked. A transition may have several firing modes, each with its own rate and arc weights.

Petri nets and queueing models are two notations for the same processes, and two of the scripts make that explicit by building the same system both ways and comparing the generators. Because the firing time need not be exponential, the family also carries the non-Markovian cases.

The figures of this section are drawn in the notation of the net rather than in that of a queueing network. A place is a circle, and the tokens or the number inside it are the marking the model starts from; a transition is a bar, hollow when it fires after a random delay and solid when it fires at once; an arc is plain when it moves one token and labelled with its multiplicity otherwise, and an arc ending in a small circle is an inhibitor, which holds its transition disabled for as long as its place is marked. A transition with more than one firing mode is named with the number of modes, since no single distribution describes it. A circle bisected by a bar is a queueing place, whose tokens are served by a scheduling station embedded in the place.

2.10.1 `spn_basic_closed`

The closed counterpart, a token circulating between two places.

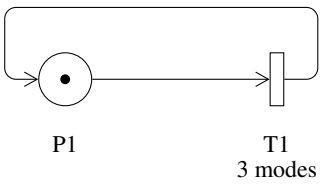

Figure 2.90: Topology of `spn_basic_closed`: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; a transition offering several firing modes is named with their number in place of a distribution.

Nodes	2: Place, Transition
Classes	1: Class1 (closed, $N = 1$)
Features	1 transition with 3 firing modes each
Solvers	JMT

Table 2.95: Features of `spn_basic_closed`.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
P1 = Place(model, 'P1')
```

```
T1 = Transition(model, 'T1')
jobclass = ClosedClass(model, 'Class1', 1, P1)
# T1 - Mode 1: Exponential
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp.fit_mean(1.0))
T1.set_enabling_conditions(mode1, jobclass, P1, 1)
T1.set_firing_outcome(mode1, jobclass, P1, 1)
# T1 - Mode 2: Erlang
mode2 = T1.add_mode('Mode2')
T1.set_distribution(mode2, Erlang.fit_mean_and_order(1.0, 2))
T1.set_enabling_conditions(mode2, jobclass, P1, 1)
T1.set_firing_outcome(mode2, jobclass, P1, 1)
# T1 - Mode 3: HyperExp
mode3 = T1.add_mode('Mode3')
T1.set_distribution(mode3, HyperExp.fit_mean_and_scv(1.0, 4.0))
T1.set_enabling_conditions(mode3, jobclass, P1, 1)
T1.set_firing_outcome(mode3, jobclass, P1, 1)
# Routing
R = model.init_routing_matrix()
R.set(jobclass, jobclass, P1, T1, 1.0)
R.set(jobclass, jobclass, T1, P1, 1.0)
model.link(R)
# Set initial state

    ... (3 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass  QLen  Util    RespT   ResidT   ArvR    Tput
P1       Class1       1     0  0.35308  0.35308  2.835  2.8322
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.2 `spn_basic_open`

One place and one timed transition fed by an open class, the Petri net form of an M/M/1 queue.

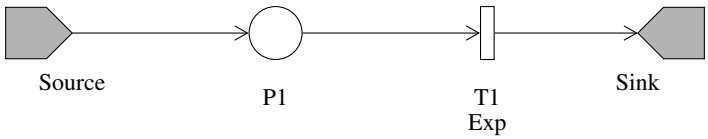

Figure 2.91: Topology of `spn_basic_open`: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
P1 = Place(model, 'P1')
```

Nodes	4: Source, Place, Transition, Sink
Classes	1: Class1 (open)
Arrivals	Source – Class1: Exp(1)
Features	1 transition with 1 firing mode each
Solvers	JMT

Table 2.96: Features of `spn_basic_open`.

```
T1 = Transition(model, 'T1')
jobclass = OpenClass(model, 'Class1', 0)
source.set_arrival(jobclass, Exp(1.0))
# T1
mode = T1.add_mode('Mode1')
T1.set_number_of_servers(mode, GlobalConstants.MaxInt)
T1.set_distribution(mode, Exp(4.0))
T1.set_enabling_conditions(mode, jobclass, P1, 1)
T1.set_firing_outcome(mode, jobclass, sink, 1)
# Routing
R = model.init_routing_matrix()
R.set(jobclass, jobclass, source, P1, 1.0)
R.set(jobclass, jobclass, P1, T1, 1.0)
R.set(jobclass, jobclass, T1, sink, 1.0)
model.link(R)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)

model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen  Util   RespT  ResidT    ArvR    Tput
Source   Class1          0     0       0       0       0  1.0312
P1       Class1    0.25966     0  0.2503  0.2503  1.0312  1.0427
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.3 `spn_closed_fourplaces`

Four places with Erlang, Coxian and hyperexponential firing times.

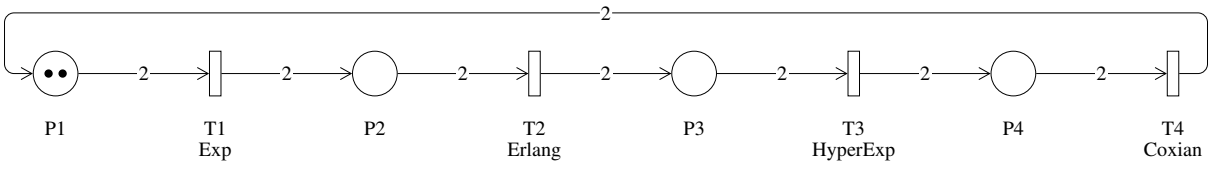

Figure 2.92: Topology of `spn_closed_fourplaces`: a stochastic Petri net over 4 places and 4 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
```

Nodes	8: Place × 4, Transition × 4
Classes	1: Class1 (closed, $N = 2$)
Features	4 transitions with 1 firing mode each
Solvers	JMT

Table 2.97: Features of `spn_closed_fourplaces`.

```
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
P3 = Place(model, 'P3')
P4 = Place(model, 'P4')
T1 = Transition(model, 'T1')
T2 = Transition(model, 'T2')
T3 = Transition(model, 'T3')
T4 = Transition(model, 'T4')
jobclass = ClosedClass(model, 'Class1', 2, P1, 0)
# T1 - Exponential distribution
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp(2.0))
T1.set_enabling_conditions(mode1, jobclass, P1, 2)
T1.set_firing_outcome(mode1, jobclass, P2, 2)
# T2 - Erlang distribution
mode2 = T2.add_mode('Mode2')
T2.set_distribution(mode2, Erlang(3, 4))
T2.set_enabling_conditions(mode2, jobclass, P2, 2)
T2.set_firing_outcome(mode2, jobclass, P3, 2)
# T3 - HyperExp distribution
mode3 = T3.add_mode('Mode3')
T3.set_distribution(mode3, HyperExp(0.7, 3, 1.5))
T3.set_enabling_conditions(mode3, jobclass, P3, 2)
T3.set_firing_outcome(mode3, jobclass, P4, 2)

    ... (25 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen  Util    RespT   ResidT     ArvR     Tput
P1       Class1    0.27748     0  0.48217  0.48217  0.57461  0.57432
P2       Class1    0.77676     0   1.3441   1.3441  0.57432  0.57273
P3       Class1    0.25164     0  0.43035  0.43035  0.57273  0.57294
P4       Class1    0.69386     0   1.2301   1.2301  0.57294   0.5744
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.4 `spn_closed_twoplaces`

A closed net over two places with Erlang firing times.
The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
T1 = Transition(model, 'T1')
```

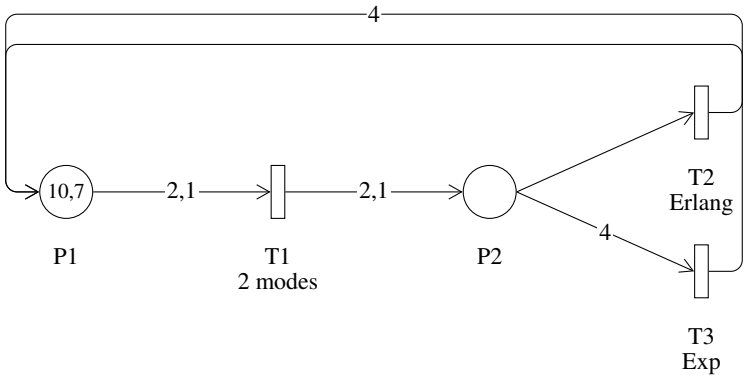

Figure 2.93: Topology of `spn_closed_twoplaces`: a stochastic Petri net over 2 places and 3 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity; a transition offering several firing modes is named with their number in place of a distribution.

Nodes	5: Place × 2, Transition × 3
Classes	2: Class1 (closed, $N = 10$), Class2 (closed, $N = 7$)
Features	3 transitions, 1 to 2 firing modes; class switching on 2 links
Solvers	JMT

Table 2.98: Features of `spn_closed_twoplaces`.

```
T2 = Transition(model, 'T2')
T3 = Transition(model, 'T3')
jobclass1 = ClosedClass(model, 'Class1', 10, P1, 0)
jobclass2 = ClosedClass(model, 'Class2', 7, P1, 0)
# T1 - Mode 1: Class1, requires 2 tokens, produces 2 tokens
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp(2.0))
T1.set_enabling_conditions(mode1, jobclass1, P1, 2)
T1.set_firing_outcome(mode1, jobclass1, P2, 2)
# T1 - Mode 2: Class2, requires 1 token, produces 1 token
mode2 = T1.add_mode('Mode2')
T1.set_distribution(mode2, Exp(3.0))
T1.set_enabling_conditions(mode2, jobclass2, P1, 1)
T1.set_firing_outcome(mode2, jobclass2, P2, 1)
# T2 - Mode 3: Class1 with Erlang
mode3 = T2.add_mode('Mode3')
T2.set_distribution(mode3, Erlang(1.5, 2))
T2.set_enabling_conditions(mode3, jobclass1, P2, 1)
T2.set_firing_outcome(mode3, jobclass1, P1, 1)
# T3 - Mode 4: Class2, requires 4 tokens, produces 4 tokens
mode4 = T3.add_mode('Mode4')

    ... (19 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen  Util   RespT  ResidT     ArvR     Tput
P1       Class1    0.88109     0  1.1688  1.1688   0.7533  0.75488
```

```
P1       Class2      1.708     0  1.1162  1.1162   1.5906   1.6042
P2       Class1     9.1189     0    12.1    12.1  0.75488  0.75495
P2       Class2      5.292     0  3.3423  3.3423   1.6042   1.5821
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by JMT.

2.10.5 spn_fluid_dae

Fluid DAE analysis of stochastic Petri nets, including invariant constraints, immediate-transition flows, bounded markings and marking covariances.

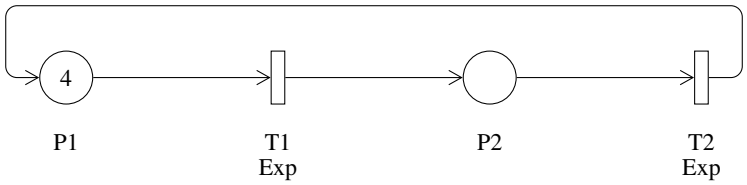

Figure 2.94: Topology of spn_fluid_dae: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once.

Nodes	4: Place × 2, Transition × 2
Classes	1: Class1 (closed, $N = 4$)
Features	2 transitions with 1 firing mode each
Solvers	FLD, CTMC

Table 2.99: Features of spn_fluid_dae.

The example is built and solved as follows.

```
tolerance -- and supply the rank the drift Jacobian is missing;
the constraint that its input place holds no mass;
"""A closed net whose fluid answer is EXACT.
Every mode is infinite-server with one input arc, so min(m/w, Inf) = m and
the drift is LINEAR: the fluid mean is then the exact mean, and the
covariance the exact covariance (a binomial marking).
"""
exact = Network('spn_fluid_exact')
P1 = Place(exact, 'P1')
P2 = Place(exact, 'P2')
T1 = Transition(exact, 'T1')
T2 = Transition(exact, 'T2')
jc = ClosedClass(exact, 'Class1', 4, P1, 0)
m1 = T1.add_mode('Mode1')
T1.set_number_of_servers(m1, float('inf'))
T1.set_distribution(m1, Exp(2))
T1.set_enabling_conditions(m1, jc, P1, 1)
T1.set_firing_outcome(m1, jc, P2, 1)
m2 = T2.add_mode('Mode2')
T2.set_number_of_servers(m2, float('inf'))
T2.set_distribution(m2, Exp(3))
T2.set_enabling_conditions(m2, jc, P2, 1)
T2.set_firing_outcome(m2, jc, P1, 1)
R = exact.init_routing_matrix()
R.set(jc, jc, P1, T1, 1.0)
R.set(jc, jc, T1, P2, 1.0)

    ... (83 source lines omitted; full implementation: python/examples/basic/)

    petri['invariantLabel'][0],
    petri['invariantValue'][0],
```

```
    petri['invariantError'][0]))

print(FLD(spn_fluid_immediate()).getAvgTable())
print(FLD(spn_fluid_reduced()).getAvgTable())
```

Running this edition prints

```
Fluid analysis [method: default/dae; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station  JobClass  QLen  Util    RespT   ResidT  ArvR  Tput
P1       Class1     2.4   2.4      0.5      0.5   4.8   4.8
P2       Class1     1.6   1.6  0.33333  0.33333   4.8   4.8
  Stochcomp: 120 immediate states eliminated, 5 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
Station  JobClass  QLen  Util    RespT   ResidT  ArvR  Tput
P1       Class1     2.4   2.4      0.5      0.5   4.8   4.8
P2       Class1     1.6   1.6  0.33333  0.33333   4.8   4.8
marking variance: 0.960000 0.960000  (exact 0.96)
invariant "P1(class 1) + P2(class 1)" = 4, error -4.44e-16
Fluid analysis [method: default/dae; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station  JobClass   QLen   Util    RespT   ResidT    ArvR    Tput
P1       Class1    2.753  2.753   1.4273   1.4273  1.9288  1.9288
P2       Class1        0      0        0        0  1.9288  1.9288
P3       Class1    1.247  1.247  0.64652  0.64652  1.9288  1.9288
Fluid analysis [method: default/dae; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station  JobClass   QLen   Util    RespT   ResidT    ArvR    Tput
P1       Class1    2.753  2.753   1.4273   1.4273  1.9288  1.9288
P3       Class1    1.247  1.247  0.64652  0.64652  1.9288  1.9288
```

The rows repeat for each of the 2 solvers the script runs (FLD, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.10.6 `spn_fourmodes`

Four firing modes, and the mode selection probabilities.

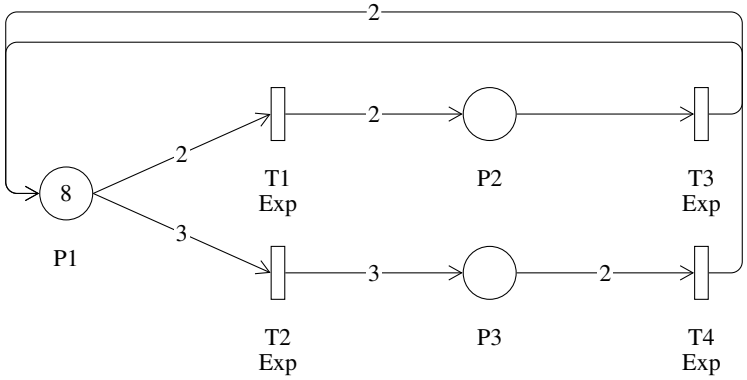

Figure 2.95: Topology of `spn_fourmodes`: a stochastic Petri net over 3 places and 4 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

Nodes	7: Place × 3, Transition × 4
Classes	1: Class1 (closed, $N = 8$)
Features	4 transitions with 1 firing mode each
Solvers	JMT

Table 2.100: Features of `spn_fourmodes`.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
P3 = Place(model, 'P3')
T1 = Transition(model, 'T1')
T2 = Transition(model, 'T2')
T3 = Transition(model, 'T3')
T4 = Transition(model, 'T4')
jobclass = ClosedClass(model, 'Class1', 8, P1, 0)
# T1 - Mode 1: P1 -> P2 (2 tokens)
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp(2.0))
T1.set_enabling_conditions(mode1, jobclass, P1, 2)
T1.set_firing_outcome(mode1, jobclass, P2, 2)
# T2 - Mode 2: P1 -> P3 (3 tokens)
mode2 = T2.add_mode('Mode2')
T2.set_distribution(mode2, Exp(1.0))
T2.set_enabling_conditions(mode2, jobclass, P1, 3)
T2.set_firing_outcome(mode2, jobclass, P3, 3)
# T3 - Mode 3: P2 -> P1 (1 token)
mode3 = T3.add_mode('Mode3')
T3.set_distribution(mode3, Exp(4.0))
T3.set_enabling_conditions(mode3, jobclass, P2, 1)
T3.set_firing_outcome(mode3, jobclass, P1, 1)
# T4 - Mode 4: P3 -> P1 (2 tokens)

    ... (21 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen  Util    RespT   ResidT    ArvR    Tput
P1       Class1    3.4344     0  0.72032  0.72032   4.796  4.7866
P2       Class1    2.5589     0  0.86649  0.86649  2.9984  3.0006
P3       Class1    1.9935     0   1.1405   1.1405  1.8044   1.742
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.7 `spn_inhibiting`

An inhibitor arc, which disables a transition while its place is marked.

Nodes	6: Place × 3, Transition × 3
Classes	1: Class1 (closed, $N = 4$)
Features	3 transitions, 1 to 2 firing modes
Solvers	JMT

Table 2.101: Features of `spn_inhibiting`.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
```

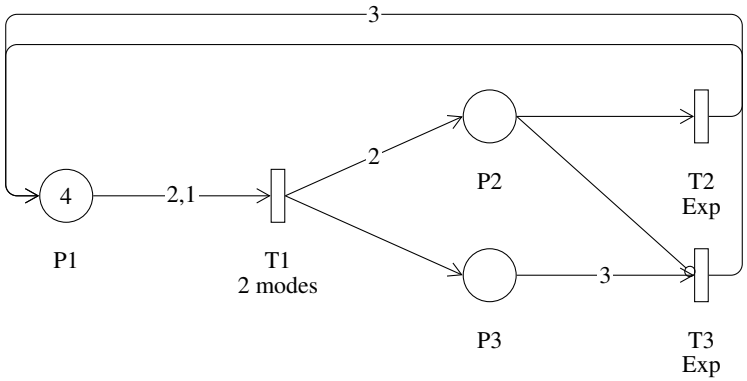

Figure 2.96: Topology of `spn_inhibiting`: a stochastic Petri net over 3 places and 3 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity; the arc ending in a circle is an inhibitor and holds T3 disabled while P2 is marked; a transition offering several firing modes is named with their number in place of a distribution.

```
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
P3 = Place(model, 'P3')
T1 = Transition(model, 'T1')
T2 = Transition(model, 'T2')
T3 = Transition(model, 'T3')
jobclass = ClosedClass(model, 'Class1', 4, P1, 0)
# T1 - Mode 1: requires 2 tokens from P1, produces 2 tokens to P2
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp(2.0))
T1.set_enabling_conditions(mode1, jobclass, P1, 2)
T1.set_firing_outcome(mode1, jobclass, P2, 2)
# T1 - Mode 2: requires 1 token from P1, produces 1 token to P3
mode2 = T1.add_mode('Mode2')
T1.set_distribution(mode2, Exp(1.0))
T1.set_enabling_conditions(mode2, jobclass, P1, 1)
T1.set_firing_outcome(mode2, jobclass, P3, 1)
# T2 - Mode 3: P2 -> P1
mode3 = T2.add_mode('Mode3')
T2.set_distribution(mode3, Exp(4.0))
T2.set_enabling_conditions(mode3, jobclass, P2, 1)
T2.set_firing_outcome(mode3, jobclass, P1, 1)
# T3 - Mode 4: P3 -> P1 with inhibiting condition on P2
mode4 = T3.add_mode('Mode4')

    ... (21 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen  Util    RespT   ResidT    ArvR     Tput
P1       Class1     1.3667     0  0.56802  0.56802  2.4787   2.4778
P2       Class1    0.74496     0  0.43709  0.10927  1.7363   1.7134
P3       Class1     1.8403     0   2.6351   1.3176  0.7158  0.71634
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and

throughput of each station-class pair, as returned by `JMT`.

2.10.8 spn_lpbounds

Linear-programming bounds on a Petri net with SolverBA: the stationary chain is relaxed to a moment polytope and every reported measure minimised and maximised over it, reproducing Table 2 of Liu (1998) on a blocking production line and then bracketing the exact marking of an inhibitor net.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	CTMC

Table 2.102: Features of `spn_lpbounds`.

The example is built and solved as follows.

```
SolverBA's 'spnlp' family is the first BOUNDING route LINE offers for a Petri
net. SolverCTMC builds the explicit generator, SolverSSA and SolverLDES
simulate, SolverFLD fluidises, and SolverNC 'rec' needs a product form; this one
needs none of that. It relaxes the stationary chain to a MOMENT POLYTOPE -- the
uniformized evolution equation written for E[X_p], E[X_p^2] and E[X_p1 X_p2],
plus behavioural and probabilistic inequalities -- and then minimises and
maximises each reported measure over it. Every stationary point of the true
chain satisfies every row, so the two optima bracket the exact value.
    spn_lpbnd        assembles the polytope and solves the LPs
    spn_sinvariants  supplies the weighted place invariants, which are both the
                     conservation equalities and the a priori per-place caps
Reference: Z. Liu, "Performance Analysis of Stochastic Timed Petri Nets Using
Linear Programming Approach", IEEE Trans. Software Engineering 24(11), 1998,
1014-1030.
                        Transition)
def prodline(mu):
    """The reference's Fig. 2b: four servers, blocking before service.
    Server i cannot start until the downstream buffer has a free slot. The
    buffers hold 3, 2 and 4, and each is a conserved pair of places --
    (p5,p2), (p4,p1), (p3,p0) -- so the net is a strongly connected marked graph
    and all four transitions carry the same throughput.
    """
    model = Network('liu98')
    p5, p4, p3 = Place(model, 'p5'), Place(model, 'p4'), Place(model, 'p3')
    p2, p1, p0 = Place(model, 'p2'), Place(model, 'p1'), Place(model, 'p0')
    t1, t2 = Transition(model, 't1'), Transition(model, 't2')

    ... (110 source lines omitted; full implementation: python/examples/basic/)

          % (ex.Station.iloc[i], lo.QLen.iloc[i], ex.QLen.iloc[i], up.QLen.iloc[i]))
# U = Q at a Place, which LINE models as an INF station -- the same convention
# SolverCTMC and SolverNC report. The paper's place utilization 1 - P(m = 0) is
# a different quantity and is not this column.
print('\nsolver.citations():')
SolverBA(inhibiting(), 'spnlp.upper').citations(display=True)
```

Running this edition prints

```
Liu (1998) Table 2: total throughput of the production line
case  Markovian LP                    published          operational LP
          lower     simul     upper       l.b.     u.b.2      o.l.b.    o.u.b.
1        1.1653    1.9510    2.0000      1.165     2.000      0.9302    2.0000
2        1.8288    2.9780    3.5294      1.829     3.529      1.4815    4.0000
3        1.5814    2.8730    3.3333      1.581     3.333      1.3333    4.0000
4        1.3592    2.7570    3.3333      1.359     3.333      1.1111    4.0000
5        1.3497    2.6670    2.9627      1.350     2.963      1.1110    4.4440

methods offered on this net: spnlp.lower, spnlp.op.lower, spnlp.op.upper, spnlp.upper
  Stochcomp: 225 immediate states eliminated, 15 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
```

```
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

Mean tokens per place, exact between the two sides:
place         lower      exact      upper
P1          0.28916    1.40270    2.45455
P2          0.11111    0.75000    2.15385
P3          0.53333    1.84730    3.20482

solver.citations():
Z. Liu, "Performance Analysis of Stochastic Timed Petri Nets Using Linear Programming Approach", IEEE
    Trans. Software Engineering 24(11), 1998
    covers: uniformized moment relaxation of the marking process, and the linear program bounding the mean
     tokens and the throughputs over it
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

2.10.9 `spn_open_sevenplaces`

A seven-place open net, large enough that the state space matters.

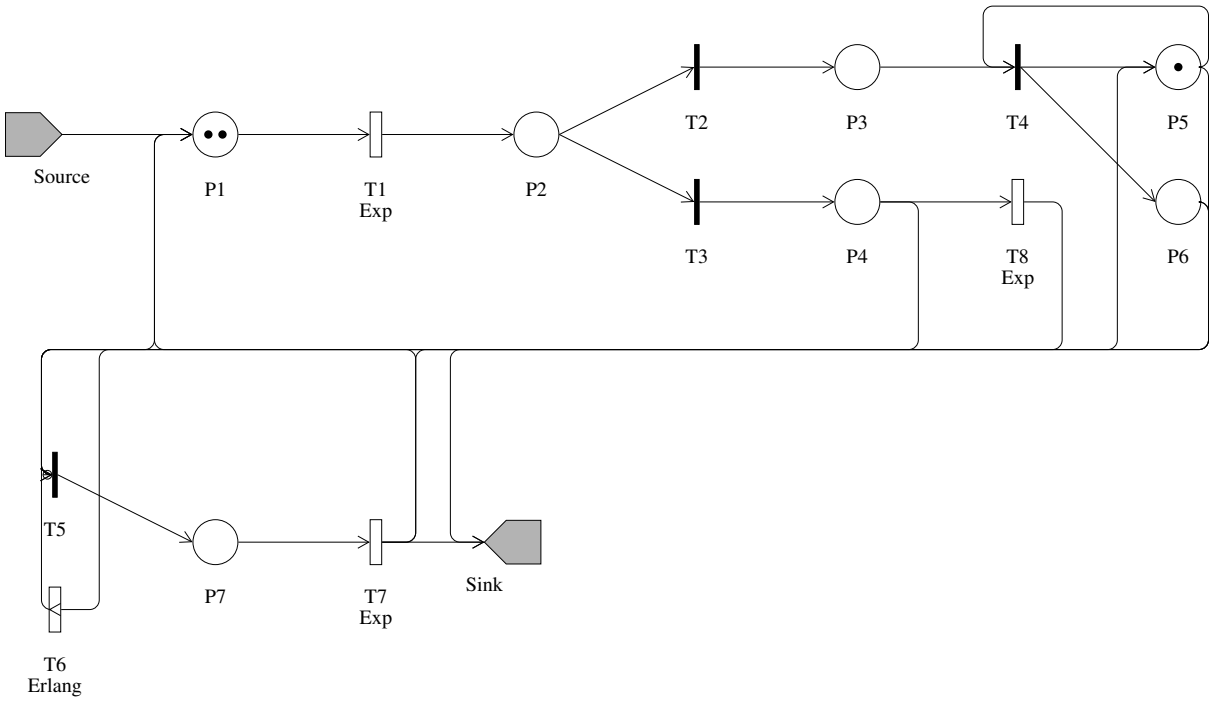

Figure 2.97: Topology of `spn_open_sevenplaces`: a stochastic Petri net over 7 places and 8 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the arc ending in a circle is an inhibitor and holds T5 disabled while P6 is marked; the net is open, fed by a Source and drained into a Sink.

Nodes	17: Source, Place × 7, Transition × 8, Sink
Classes	1: Class1 (open)
Arrivals	Source – Class1: Exp(1)
Features	8 transitions with 1 firing mode each
Solvers	JMT

Table 2.103: Features of `spn_open_sevenplaces`.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
# Create 7 places
P = [Place(model, f'P{i+1}') for i in range(7)]
# Create 8 transitions
T = [Transition(model, f'T{i+1}') for i in range(8)]
jobclass = OpenClass(model, 'Class1', 0)
source.set_arrival(jobclass, Exp.fit_mean(1.0))
# T1 - Timed transition with exponential distribution
mode1 = T[0].add_mode('Mode1')
T[0].set_number_of_servers(mode1, GlobalConstants.MaxInt)
T[0].set_distribution(mode1, Exp(4.0))
T[0].set_enabling_conditions(mode1, jobclass, P[0], 1)
T[0].set_firing_outcome(mode1, jobclass, P[1], 1)
# T2 - Immediate transition
mode2 = T[1].add_mode('Mode1')
T[1].set_number_of_servers(mode2, GlobalConstants.MaxInt)
T[1].set_enabling_conditions(mode2, jobclass, P[1], 1)
T[1].set_firing_outcome(mode2, jobclass, P[2], 1)
T[1].setTimingStrategy(mode2, TimingStrategy.IMMEDIATE)
T[1].setFiringPriorities(mode2, 1)
T[1].setFiringWeights(mode2, 1.0)
# T3 - Immediate transition
mode3 = T[2].add_mode('Mode1')

    ... (81 source lines omitted; full implementation: python/examples/basic/)


model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen  Util    RespT    ResidT     ArvR     Tput
Source   Class1          0     0        0         0        0   1.0153
P1       Class1    0.73504     0  0.25337   0.48798   2.8389   2.8376
P2       Class1          0     0        0         0   2.8376   2.8376
P3       Class1    0.14404     0  0.10279  0.098981   1.4098   1.4058
P4       Class1    0.50366     0  0.35917   0.17293   1.4223   1.4232
P5       Class1    0.79047     0  0.43293   0.28862   1.8031   1.8066
P6       Class1     1.3832     0  0.98965   0.40319   1.4058   1.4006
P7       Class1     0.2102     0  0.50077   0.77897  0.41759  0.41932
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.10 `spn_pareto_service`

A Pareto firing time, a heavy-tailed non-Markovian net.

Nodes	4: Source, Place, Transition, Sink
Classes	1: Class1 (open)
Arrivals	Source – Class1: Exp(0.5)
Features	1 transition with 1 firing mode each
Solvers	JMT

Table 2.104: Features of `spn_pareto_service`.

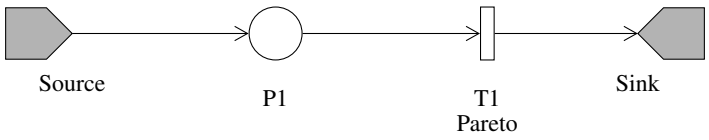

Figure 2.98: Topology of `spn_pareto_service`: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

The example is built and solved as follows.

```
GlobalConstants.setVerbose(VerboseLevel.STD)
model = Network("model")
# Nodes
source = Source(model, "Source")
sink = Sink(model, "Sink")
P1 = Place(model, "P1")
T1 = Transition(model, "T1")
# Source arrival
jobclass = OpenClass(model, "Class1")
source.setArrival(jobclass, Exp(0.5))  # arrival rate 0.5
# T1 with Pareto service time
# Pareto(shape, scale) - shape must be >= 2
# With shape=3 and scale=1, mean service time = 3*1/(3-1) = 1.5
mode = T1.addMode("Mode1")
T1.setNumberOfServers(mode, 1)
T1.setDistribution(mode, Pareto(3, 1))  # Pareto with shape=3, scale=1
T1.setEnablingConditions(mode, jobclass, P1, 1)
T1.setFiringOutcome(mode, jobclass, sink, 1)
# Routing
R = model.init_routing_matrix()
R.set(jobclass, jobclass, source, P1, 1.0)
R.set(jobclass, jobclass, P1, T1, 1.0)
R.set(jobclass, jobclass, T1, sink, 1.0)
model.link(R)
return model
model = spn_pareto_service()

# Solver
solver = JMT(model, seed=23000, samples=10000)
table = solver.avg_table()
print(table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen  Util  RespT  ResidT     ArvR     Tput
Source   Class1         0     0      0       0        0  0.51382
P1       Class1    2.7513     0  5.007   5.007  0.51382  0.51422
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.11 `spn_productform_nc`

Product-form analysis of cyclic and fork–join stochastic Petri nets with SolverNC, compared against an explicit CTMC and accompanied by the structural certificate and marking invariants.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	NC, CTMC

Table 2.105: Features of `spn_productform_nc`.

The example is built and solved as follows.

```
Solve a stochastic Petri net analytically with SolverNC.
SolverNC's 'rec' method is the first ANALYTICAL route LINE offers for a Petri
net: SolverCTMC builds the explicit generator, SolverSSA and SolverLDES
simulate, SolverFLD fluidises. It works in three steps, each with its own
reference:
    spn_pf      decides whether the net has a product form and derives the
                per-place factors g_l, by complex balance
                (Coleman-Henderson-Taylor, Perform. Eval. 26(3), 1996)
    mdd_rec     evaluates G = sum over the reachable set of prod_l g_l(m_l) by
                ONE memoised walk of the decision diagram holding that set
                (Balsamo-Marin-Stojic, FGCS 111 (2020) 475-490)
    spn_metrics reads the mean tokens, the utilisations and the throughputs off
                masked walks of the same diagram
Two nets are solved here. The first is a closed cycle, which a queueing network
could also express. The second FORKS: its transition Tf consumes one token and
produces two, and Tj consumes two and produces one, so the marking is not a
conserved job population and there is no queueing-network counterpart -- which
is exactly the limitation the MDD-rec paper opens with.
def cyclic(ntokens=4):
    rates = [1.0, 1.5, 2.0]
    model = Network('spn')
    places = [Place(model, 'P%d' % i) for i in range(3)]
    trans = [Transition(model, 'T%d' % i) for i in range(3)]
    cls = ClosedClass(model, 'Class1', ntokens, places[0])
    for i in range(3):
        mode = trans[i].add_mode('fire')

    ... (64 source lines omitted; full implementation: python/examples/basic/)

# one throughput -- a flow-conservation law nothing in the derivation was told.
print('mode throughputs: %s (they must all agree)' % np.round(met['modeTput'], 8))
# The place invariant of this net is 2*m0 + m1 + m2 + 2*m3 = 6, not the token
# count, which is what "not a conserved population" means concretely.
print('place invariant 2*m0 + m1 + m2 + 2*m3 = %.6f'
      % float(np.array([2, 1, 1, 2]) @ np.asarray(met['tokens'])))
```

Running this edition prints

```
--- 3-place cyclic net, N = 4 ---
NC analysis [method: rec; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
P0       Class1     2.2499   2.2499   2.5884   2.5884  0.8692  0.8692
P1       Class1     1.0698   1.0698   1.2308   1.2308  0.8692  0.8692
P2       Class1    0.68029  0.68029  0.78266  0.78266  0.8692  0.8692
The same net through the explicit generator, for comparison:
  Stochcomp: 105 immediate states eliminated, 15 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
P0       Class1     2.2499   2.2499   2.5884   2.5884  0.8692  0.8692
P1       Class1     1.0698   1.0698   1.2308   1.2308  0.8692  0.8692
P2       Class1    0.68029  0.68029  0.78266  0.78266  0.8692  0.8692

SPN product form: geometric, 3 complexes, 1 linkage classes, rank 2, deficiency 0, weakly reversible

... (4 captured-output lines omitted; rerun the source example for the full output)

NC analysis [method: rec; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
P0       C         0.71026  0.71026   1.1694   1.1694  0.60736  0.60736
P1       C          1.8642   1.8642   3.0693   1.5347  0.60736  0.60736
P2       C          1.8642   1.8642   3.0693   1.5347  0.60736  0.60736
P3       C         0.42554  0.42554  0.70064  0.70064  0.60736  0.60736
mode throughputs: [0.60736088 0.60736088 0.60736088] (they must all agree)
place invariant 2*m0 + m1 + m2 + 2*m3 = 6.000000
```

The rows repeat for each of the 2 solvers the script runs (NC, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

2.10.12 spn_queueing_place

A queueing station and a Petri net in one model, the two formalisms side by side.

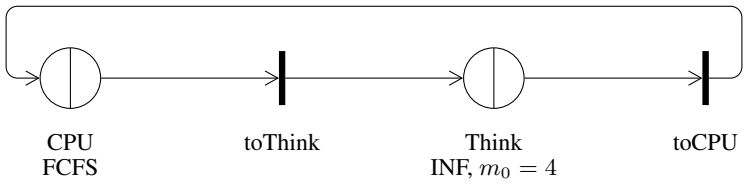

Figure 2.99: Topology of spn_queueing_place: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the 2 bisected circles are queueing places, whose tokens are served by an embedded station.

Nodes	4: Place × 2, Transition × 2
Classes	1: Jobs (closed, $N = 4$)
Service	CPU – Jobs: Exp(1.5) Think – Jobs: Exp(0.5)
Features	2 transitions with 1 firing mode each
Solvers	MVA, LDES

Table 2.106: Features of spn_queueing_place.

The example is built and solved as follows.

```
A queueing place embeds a scheduling station inside a Petri-net place: an
arriving token is served by the place's embedded queue and, on completion,
moves to a depository from which the output transitions consume it.
Here a CPU (single-server FCFS queueing place) and a think stage (infinite-
server queueing place) exchange a fixed population of N tokens through two
immediate transitions. This is the queueing-Petri-net rendering of a machine-
repairman / finite-population model, so its exact solution is available from
SolverMVA on the equivalent Delay+Queue network for cross-validation.
N = 4
model = Network('QueueingPetriNet')
# Two queueing places: set_service turns each Place into a queueing place whose
# embedded queue is served under the scheduling strategy of its constructor.
cpu = Place(model, 'CPU', SchedStrategy.FCFS)     # single-server embedded queue
think = Place(model, 'Think', SchedStrategy.INF)  # infinite-server embedded queue
jobs = ClosedClass(model, 'Jobs', N, think, 0)
cpu.setService(jobs, Exp(1.5))    # embedded FCFS service, rate 1.5
think.setService(jobs, Exp(0.5))  # embedded think time, mean 2
# Two immediate transitions cycle one token per firing between the places.
toCPU = Transition(model, 'toCPU')
m1 = toCPU.addMode('m1')
toCPU.setTimingStrategy(m1, TimingStrategy.IMMEDIATE)
toCPU.setEnablingConditions(m1, jobs, think, 1)
toCPU.setFiringOutcome(m1, jobs, cpu, 1)
toThink = Transition(model, 'toThink')
m2 = toThink.addMode('m2')
toThink.setTimingStrategy(m2, TimingStrategy.IMMEDIATE)

    ... (16 source lines omitted; full implementation: python/examples/basic/)

queue = Queue(ref, 'CPU', SchedStrategy.FCFS)
cc = ClosedClass(ref, 'Jobs', N, delay, 0)
```

```
delay.setService(cc, Exp(0.5))
queue.setService(cc, Exp(1.5))
ref.link(Network.serialRouting(delay, queue))
print(SolverMVA(ref).avgTable())
```

Running this edition prints

```
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
Station  JobClass    QLen     Util   RespT  ResidT    ArvR    Tput
CPU      Jobs      1.6189  0.79356  1.3626  1.3626  1.1881  1.1881
Think    Jobs      2.3811   2.3811   2.004   2.004  1.1881  1.1881
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass    QLen     Util  RespT  ResidT    ArvR    Tput
Think    Jobs      2.3817   2.3817      2       2  1.1908  1.1908
CPU      Jobs      1.6183  0.79389  1.359   1.359  1.1908  1.1908
```

The rows repeat for each of the 2 solvers the script runs (MVA, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.10.13 spn_twomodes

A transition with two firing modes.

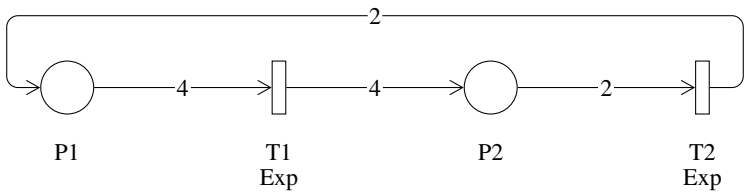

Figure 2.100: Topology of spn_twomodes: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

Nodes	4: Place × 2, Transition × 2
Classes	1: Class1 (closed, $N = 10$)
Features	2 transitions with 1 firing mode each
Solvers	JMT

Table 2.107: Features of spn_twomodes.

The example is built and solved as follows.

```
"""Build and return the SPN model."""
model = Network('model')
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
T1 = Transition(model, 'T1')
T2 = Transition(model, 'T2')
jobclass = ClosedClass(model, 'Class1', 10, P1, 0)
# T1: requires 4 tokens from P1, produces 4 tokens to P2
mode1 = T1.add_mode('Mode1')
T1.set_distribution(mode1, Exp(2.0))
T1.set_enabling_conditions(mode1, jobclass, P1, 4)
T1.set_firing_outcome(mode1, jobclass, P2, 4)
# T2: requires 2 tokens from P2, produces 2 tokens to P1
mode2 = T2.add_mode('Mode2')
T2.set_distribution(mode2, Exp(3.0))
T2.set_enabling_conditions(mode2, jobclass, P2, 2)
T2.set_firing_outcome(mode2, jobclass, P1, 2)
# Routing
R = model.init_routing_matrix()
```

```
R.set(jobclass, jobclass, P1, T1, 1.0)
R.set(jobclass, jobclass, P2, T2, 1.0)
R.set(jobclass, jobclass, T1, P2, 1.0)
R.set(jobclass, jobclass, T2, P1, 1.0)
model.link(R)
return model
GlobalConstants.set_verbose(VerboseLevel.STD)

model = build_model()

solver = JMT(model, seed=23000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen  Util    RespT   ResidT    ArvR    Tput
P1       Class1    4.6632     0  0.89764  0.89764  5.0348  4.9991
P2       Class1    5.3368     0   1.0474   1.0474  4.9991  5.0357
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.10.14 `test_spn_nrm_open`

The open net solved by the next-reaction method, against the exact generator.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	JMT

Table 2.108: Features of `test_spn_nrm_open`.

The example is built and solved as follows.

```
                    SSA, Sink, Source, Transition, VerboseLevel)
model = Network('mm1spn')
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
P1 = Place(model, 'P1')
T1 = Transition(model, 'T1')
jobclass = OpenClass(model, 'Class1', 0)
source.setArrival(jobclass, Exp(lam))
mode = T1.addMode('Mode1')
T1.setDistribution(mode, Exp(mu))
T1.setEnablingConditions(mode, jobclass, P1, 1)
T1.setFiringOutcome(mode, jobclass, sink, 1)
R = model.initRoutingMatrix()
R.set(jobclass, jobclass, source, P1, 1.0)
R.set(jobclass, jobclass, P1, T1, 1.0)
R.set(jobclass, jobclass, T1, sink, 1.0)
model.link(R)
return model
model = Network('tandemspn')
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
P1 = Place(model, 'P1')
P2 = Place(model, 'P2')
T1 = Transition(model, 'T1')
T2 = Transition(model, 'T2')
jobclass = OpenClass(model, 'Class1', 0)
```

```
    ... (49 source lines omitted; full implementation: python/examples/basic/)

assert abs(Qn[1] - q1) / q1 < RTOL, 'net2 P1 tokens %g vs exact %g' % (Qn[1], q1)
assert abs(Qn[2] - q2) / q2 < RTOL, 'net2 P2 tokens %g vs exact %g' % (Qn[2], q2)
assert abs(Tn[1] - lam) / lam < RTOL, 'net2 P1 tput %g vs %g' % (Tn[1], lam)
assert abs(Tn[2] - lam) / lam < RTOL, 'net2 P2 tput %g vs %g' % (Tn[2], lam)

print('test_spn_nrm_open passed')
```

Running this edition prints

```
JMT Model: <tmp>
test_spn_nrm_open passed
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

2.11 Workflow models

The workflow models express the standard control-flow patterns – sequence, probabilistic branch, parallel split and join, loop – as activity graphs, and solve them for the completion time of the whole workflow rather than for station metrics.

2.11.1 `wf_aph`

Acyclic phase-type activity durations.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# Define the workflow
wf = Workflow('APHWorkflow')
# Add activities with different SCVs using APH
A = wf.addActivity('A', APH.fitMeanAndScv(1.0, 0.5))    # Low variability
B = wf.addActivity('B', APH.fitMeanAndScv(2.0, 2.0))    # High variability
C = wf.addActivity('C', APH.fitMeanAndScv(1.5, 1.0))    # Standard variability
# Define serial precedence
wf.addPrecedence(Workflow.Serial(A, B))
wf.addPrecedence(Workflow.Serial(B, C))
return wf
wf = wf_aph()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('APH Workflow: A -> B -> C')
print('Activity A: APH(mean=1.0, SCV=0.5)')
print('Activity B: APH(mean=2.0, SCV=2.0)')
print('Activity C: APH(mean=1.5, SCV=1.0)')
print(f'Expected total mean: {1.0 + 2.0 + 1.5:.2f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Computed PH SCV: {wf.getSCV():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
APH Workflow: A -> B -> C
Activity A: APH(mean=1.0, SCV=0.5)
Activity B: APH(mean=2.0, SCV=2.0)
Activity C: APH(mean=1.5, SCV=1.0)
Expected total mean: 4.50
```

```
Computed PH mean: 4.5000
Computed PH SCV: 0.5309
Number of phases: 5
```

2.11.2 wf_branch

A probabilistic branch and its merge.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
  +-> B (60%) --+
  +-> C (40%) --+
# Define the workflow
wf = Workflow('BranchingWorkflow')
# Add activities
A = wf.addActivity('A', Exp.fitMean(1.0))
B = wf.addActivity('B', Exp.fitMean(2.0))
C = wf.addActivity('C', Exp.fitMean(5.0))
D = wf.addActivity('D', Exp.fitMean(0.5))
# Define precedences with probabilities
wf.addPrecedence(Workflow.OrFork(A, [B, C], np.array([0.6, 0.4])))
wf.addPrecedence(Workflow.OrJoin([B, C], D))
return wf
wf = wf_branch()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('Branching Workflow: A -> [B(60%) | C(40%)] -> D')
print('Activity means: A=1.0, B=2.0, C=5.0, D=0.5')
expected_branch = 0.6 * 2.0 + 0.4 * 5.0
print(f'Expected branch mean: 0.6*2.0 + 0.4*5.0 = {expected_branch:.2f}')
print(f'Expected total mean: {1.0 + expected_branch + 0.5:.2f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
Branching Workflow: A -> [B(60%) | C(40%)] -> D
Activity means: A=1.0, B=2.0, C=5.0, D=0.5
Expected branch mean: 0.6*2.0 + 0.4*5.0 = 3.20
Expected total mean: 4.70
Computed PH mean: 4.7000
Number of phases: 4
```

2.11.3 wf_complex

Nested branches, forks and loops in one workflow.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
              +-> C --+
              +-> D --+
# Define the workflow
wf = Workflow('ComplexWorkflow')
# Add activities
A = wf.addActivity('A', Exp.fitMean(0.5))
B = wf.addActivity('B', Exp.fitMean(1.0))
```

```
C = wf.addActivity('C', Exp.fitMean(2.0))
D = wf.addActivity('D', Exp.fitMean(1.5))
F = wf.addActivity('F', Exp.fitMean(1.0))
G = wf.addActivity('G', Exp.fitMean(0.5))
# Define precedences
wf.addPrecedence(Workflow.Serial(A, B))        # A -> B
wf.addPrecedence(Workflow.AndFork(B, [C, D])) # B forks to C and D
wf.addPrecedence(Workflow.AndJoin([C, D], F)) # C and D join to F
wf.addPrecedence(Workflow.Serial(F, G))        # F -> G
return wf
wf = wf_complex()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('Complex Workflow: A -> B -> [C || D] -> F -> G')
print('Activity means: A=0.5, B=1.0, C=2.0, D=1.5, F=1.0, G=0.5')


    ... (12 source lines omitted; full implementation: python/examples/basic/)

print('\nSampling 10000 workflow execution times...')
samples = wf.sample(10000)
print(f'Sample mean: {np.mean(samples):.4f}')
print(f'Sample std: {np.std(samples):.4f}')
print(f'Sample min: {np.min(samples):.4f}')
print(f'Sample max: {np.max(samples):.4f}')
```

Running this edition prints

```
Complex Workflow: A -> B -> [C || D] -> F -> G
Activity means: A=0.5, B=1.0, C=2.0, D=1.5, F=1.0, G=0.5
Expected max(C,D) mean: 2.6429
Expected total mean: 5.6429
Computed PH mean: 5.6429
Number of phases: 7

Sampling 10000 workflow execution times...
Sample mean: 5.6330
Sample std: 2.5740
Sample min: 0.7041
Sample max: 34.0034
```

2.11.4 `wf_erlang`

Erlang activity durations.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# Define the workflow
wf = Workflow('ErlangWorkflow')
# Add activities with Erlang distributions (SCV < 1)
A = wf.addActivity('A', Erlang.fitMeanAndOrder(2.0, 4))  # 4 phases, SCV=0.25
B = wf.addActivity('B', Erlang.fitMeanAndOrder(3.0, 2))  # 2 phases, SCV=0.5
C = wf.addActivity('C', Exp.fitMean(1.0))                # 1 phase, SCV=1.0
# Define serial precedence
wf.addPrecedence(Workflow.Serial(A, B))
wf.addPrecedence(Workflow.Serial(B, C))
return wf
wf = wf_erlang()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
```

```
print('Erlang Workflow: A -> B -> C')
print('Activity A: Erlang(mean=2.0, phases=4), SCV=0.25')
print('Activity B: Erlang(mean=3.0, phases=2), SCV=0.50')
print('Activity C: Exp(mean=1.0), SCV=1.00')
print(f'Expected total mean: {2.0 + 3.0 + 1.0:.2f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Computed PH SCV: {wf.getSCV():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
Erlang Workflow: A -> B -> C
Activity A: Erlang(mean=2.0, phases=4), SCV=0.25
Activity B: Erlang(mean=3.0, phases=2), SCV=0.50
Activity C: Exp(mean=1.0), SCV=1.00
Expected total mean: 6.00
Computed PH mean: 6.0000
Computed PH SCV: 0.1806
Number of phases: 7
```

2.11.5 `wf_loop`

A loop with a repetition count.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# Define the workflow
wf = Workflow('LoopWorkflow')
# Add activities
A = wf.addActivity('A', Exp.fitMean(1.0))
B = wf.addActivity('B', Exp.fitMean(2.0))
C = wf.addActivity('C', Exp.fitMean(0.5))
# Define loop: A runs once, B runs 3 times, C runs once
wf.addPrecedence(Workflow.Loop(A, [B], C, count=3))
return wf
wf = wf_loop()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('Loop Workflow: A -> [B x 3] -> C')
print('Activity means: A=1.0, B=2.0, C=0.5')
print(f'Expected total mean: 1.0 + 3*2.0 + 0.5 = {1.0 + 3*2.0 + 0.5:.2f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
Loop Workflow: A -> [B x 3] -> C
Activity means: A=1.0, B=2.0, C=0.5
Expected total mean: 1.0 + 3*2.0 + 0.5 = 7.50
Computed PH mean: 7.5000
Number of phases: 3
```

2.11.6 `wf_parallel`

An AND-fork, parallel activities and a join.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
  +-> B --+
  +-> C --+
# Define the workflow
wf = Workflow('ParallelWorkflow')
# Add activities
A = wf.addActivity('A', Exp.fitMean(1.0))
B = wf.addActivity('B', Exp.fitMean(2.0))
C = wf.addActivity('C', Exp.fitMean(3.0))
D = wf.addActivity('D', Exp.fitMean(0.5))
# Define precedences
wf.addPrecedence(Workflow.AndFork(A, [B, C]))
wf.addPrecedence(Workflow.AndJoin([B, C], D))
return wf
wf = wf_parallel()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('Parallel Workflow: A -> [B || C] -> D')
print('Activity means: A=1.0, B=2.0, C=3.0, D=0.5')

# Expected max of two exponentials: E[max(X,Y)] = 1/lambda1 + 1/lambda2 - 1/(lambda1+lambda2)
lambda_B = 0.5   # rate for mean 2.0
lambda_C = 1/3   # rate for mean 3.0
expected_max = 2.0 + 3.0 - 1/(lambda_B + lambda_C)
print(f'Expected max(B,C) mean: {expected_max:.4f}')
print(f'Expected total mean: {1.0 + expected_max + 0.5:.4f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
Parallel Workflow: A -> [B || C] -> D
Activity means: A=1.0, B=2.0, C=3.0, D=0.5
Expected max(B,C) mean: 3.8000
Expected total mean: 5.3000
Computed PH mean: 5.3000
Number of phases: 5
```

2.11.7 `wf_serial`

A sequence of activities.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
# Define the workflow
wf = Workflow('SerialWorkflow')
# Add activities with exponential service times
A = wf.addActivity('A', Exp.fitMean(1.0))
B = wf.addActivity('B', Exp.fitMean(2.0))
C = wf.addActivity('C', Exp.fitMean(1.5))
# Define serial precedence
wf.addPrecedence(Workflow.Serial(A, B))
wf.addPrecedence(Workflow.Serial(B, C))
return wf
wf = wf_serial()

# Convert to phase-type distribution
alpha, T = wf.toPH()

# Display results
print('Serial Workflow: A -> B -> C')
print('Activity means: A=1.0, B=2.0, C=1.5')
```

```
print(f'Expected total mean: {1.0 + 2.0 + 1.5:.2f}')
print(f'Computed PH mean: {wf.getMean():.4f}')
print(f'Computed PH SCV: {wf.getSCV():.4f}')
print(f'Number of phases: {T.shape[0]}')
```

Running this edition prints

```
Serial Workflow: A -> B -> C
Activity means: A=1.0, B=2.0, C=1.5
Expected total mean: 4.50
Computed PH mean: 4.5000
Computed PH SCV: 0.3580
Number of phases: 3
```

Chapter 3

Advanced models

The advanced examples cover the features that leave the product-form setting: load dependence, blocking, state-dependent routing, rewards, random environments and the disciplines that no closed form covers. They live under `python/examples/advanced/`.

Each of these features narrows the set of solvers that can be used, and the scripts say which. A model that a solver cannot represent is refused rather than approximated silently, so the feature set of a solver is discoverable from the model itself.

3.1 Load-dependent stations

A load-dependent station has a service rate that is a function of the number of jobs it holds, declared through `setLoadDependence` as a rate vector $\mu(n)$. A multiserver station is the special case $\mu(n) = \min(n, c)\mu$, and writing it out as a load-dependent station is how the scripts check the two against each other. Class dependence, declared through `setClassDependence`, lets the rate depend on the class mix rather than on the total only.

The same mechanism supports the flow-equivalent server construction: solve a subnetwork in isolation for every population, tabulate its throughput, and install that table as the rate vector of a single station that replaces it.

3.1.1 `fes_aggregation`

Aggregating a subnetwork into one station and checking the aggregate against the full model.

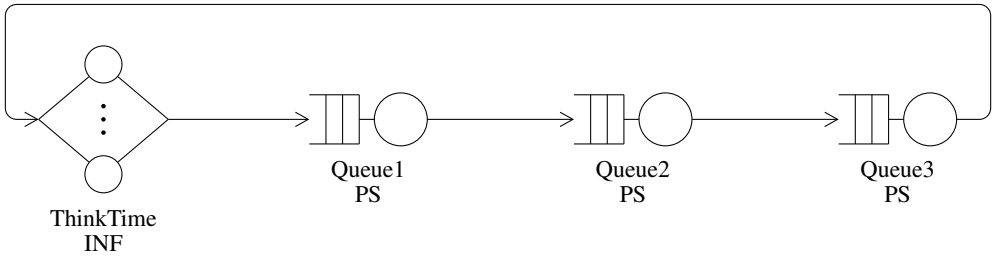

Figure 3.1: Topology of `fes_aggregation`: a closed network over 4 queueing stations.

The example is built and solved as follows.

```
Flow-Equivalent Server (FES) Aggregation Example
This example demonstrates how to use ModelAdapter.aggregate_fes to replace
a subset of stations in a closed product-form queueing network with a
single Flow-Equivalent Server (FES).
The FES has Limited Joint Dependence (LJD) service rates where the rate
```

Nodes	4: Queue × 3, Delay
Classes	2: Class1 (closed, $N = 3$), Class2 (closed, $N = 2$)
Scheduling	ThinkTime: INF; Queue1: PS; Queue2: PS; Queue3: PS
Service	ThinkTime – Class1: Exp(0.2); Class2: Exp(0.25) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.5) Queue2 – Class1: Exp(1); Class2: Exp(0.8333) Queue3 – Class1: Exp(1.25); Class2: Exp(1)
Solvers	MVA

Table 3.1: Features of `fes_aggregation`.

```
for class-c in state (n1,...,nK) equals the throughput of class-c in an
isolated subnetwork consisting only of the subset stations.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
print('=== Flow-Equivalent Server (FES) Aggregation Example ===\n')
# Create original 4-station tandem network with 2 classes
print('Creating original 4-station network...')
N1 = 3  # number of class-1 jobs
N2 = 2  # number of class-2 jobs
model = Network('OriginalModel')
# Create stations
delay = Delay(model, 'ThinkTime')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue3 = Queue(model, 'Queue3', SchedStrategy.PS)
# Create job classes
jobclass1 = ClosedClass(model, 'Class1', N1, delay, 0)
jobclass2 = ClosedClass(model, 'Class2', N2, delay, 0)
# Set service times
delay.set_service(jobclass1, Exp.fit_mean(5.0))
delay.set_service(jobclass2, Exp.fit_mean(4.0))

    ... (45 source lines omitted; full implementation: python/examples/advanced/)

print('\n--- Deaggregation Info ---')
print(f'Subset station indices: {deagg_info["subset_indices"]}')
print(f'Complement station indices: {deagg_info["complement_indices"]}')
print(f'Cutoffs used: {deagg_info["cutoffs"]}')
print(f'FES node index: {deagg_info["fes_node_idx"]}')
print('\n=== Example Complete ===')
```

Running this edition prints

```
=== Flow-Equivalent Server (FES) Aggregation Example ===

Creating original 4-station network...

--- Solving Original Model ---
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Original model results:
Station    JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
ThinkTime  Class1     1.3533   1.3533        5       5  0.27067  0.27067
ThinkTime  Class2    0.69217  0.69217        4       4  0.17304  0.17304
Queue1     Class1    0.89529    0.406   3.3077  3.3077  0.27067  0.27067
Queue1     Class2    0.72906  0.34609   4.2132  4.2132  0.17304  0.17304
Queue2     Class1    0.43467  0.27067   1.6059  1.6059  0.27067  0.27067
Queue2     Class2    0.32924  0.20765   1.9026  1.9026  0.17304  0.17304
Queue3     Class1    0.31669  0.21653     1.17    1.17  0.27067  0.27067
Queue3     Class2    0.24953  0.17304    1.442   1.442  0.17304  0.17304

... (22 captured-output lines omitted; rerun the source example for the full output)


--- Deaggregation Info ---
Subset station indices: [1 2]
Complement station indices: [0 3]
Cutoffs used: [3 2]
FES node index: 2

=== Example Complete ===
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.1.2 `fes_single_class`

The flow-equivalent construction step by step: solve the subnetwork, tabulate its throughput, install it as a load-dependent station.

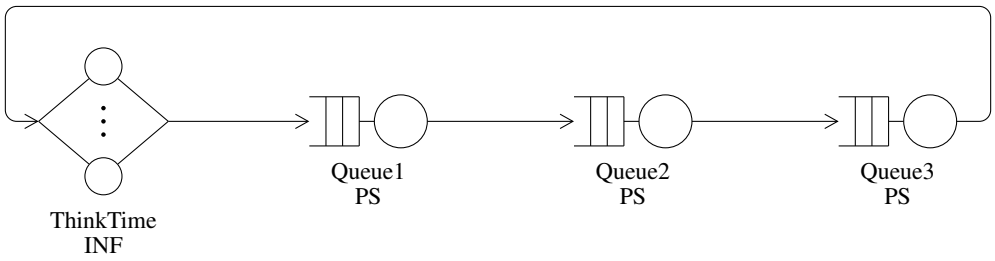

Figure 3.2: Topology of `fes_single_class`: a closed network over 4 queueing stations.

Nodes	4: Queue × 3, Delay
Classes	1: Class1 (closed, $N = 5$)
Scheduling	ThinkTime: INF; Queue1: PS; Queue2: PS; Queue3: PS
Service	ThinkTime – Class1: Exp(0.2) Queue1 – Class1: Exp(0.6667) Queue2 – Class1: Exp(1) Queue3 – Class1: Exp(1.25)
Solvers	MVA

Table 3.2: Features of `fes_single_class`.

The example is built and solved as follows.

```
Single-Class FES Aggregation Example
This tests FES aggregation with a single job class, where it should be exact.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
print('=== Single-Class FES Aggregation Example ===\n')
# Create original 4-station tandem network with 1 class
print('Creating original 4-station network...')
N1 = 5  # number of jobs
model = Network('OriginalModel')
# Create stations
delay = Delay(model, 'ThinkTime')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue3 = Queue(model, 'Queue3', SchedStrategy.PS)
# Create job class
jobclass1 = ClosedClass(model, 'Class1', N1, delay, 0)
# Set service times
delay.set_service(jobclass1, Exp.fit_mean(5.0))
queue1.set_service(jobclass1, Exp.fit_mean(1.5))
queue2.set_service(jobclass1, Exp.fit_mean(1.0))
queue3.set_service(jobclass1, Exp.fit_mean(0.8))
# Set up tandem routing
P = model.init_routing_matrix()
P[0][0] = model.serial_routing([delay, queue1, queue2, queue3])
model.link(P)
# Solve original model with MVA

    ... (37 source lines omitted; full implementation: python/examples/advanced/)

print('\n--- Deaggregation Info ---')
```

```
print(f'Subset station indices: {deagg_info["subset_indices"]}')
print(f'Complement station indices: {deagg_info["complement_indices"]}')
print(f'Cutoffs used: {deagg_info["cutoffs"]}')
print(f'FES node index: {deagg_info["fes_node_idx"]}')
print('\n=== Example Complete ===')
```

Running this edition prints

```
=== Single-Class FES Aggregation Example ===

Creating original 4-station network...

--- Solving Original Model ---
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Original model results:
Station    JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
ThinkTime  Class1     2.3333   2.3333        5       5  0.46666  0.46666
Queue1     Class1     1.4069  0.69999   3.0148  3.0148  0.46666  0.46666
Queue2     Class1    0.72959  0.46666   1.5634  1.5634  0.46666  0.46666
Queue3     Class1    0.53021  0.37333   1.1362  1.1362  0.46666  0.46666

--- Creating FES Model ---
Aggregating Queue1 and Queue2 into a single FES...


... (16 captured-output lines omitted; rerun the source example for the full output)


--- Deaggregation Info ---
Subset station indices: [1 2]
Complement station indices: [0 3]
Cutoffs used: [5]
FES node index: 2

=== Example Complete ===
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.1.3 `ld_class_dependence`

Service rates that depend on the class mix at the station, not only on the total population.

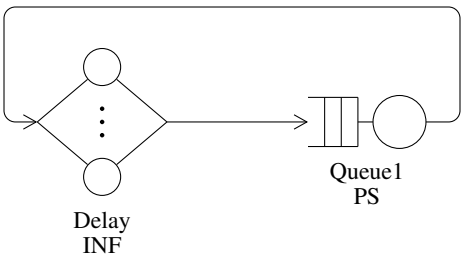

Figure 3.3: Topology of `ld_class_dependence`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 16$), Class2 (closed, $N = 8$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(0.5) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.4)
Solvers	CTMC, MVA

Table 3.3: Features of `ld_class_dependence`.

The example is built and solved as follows.

```
Load-Dependent Service - Class Dependence (product-form)
This example demonstrates class-dependent (product-form) service rates. Each
class sees a scaling beta_{i,r} that depends ONLY on its own per-class
population n_{i,r} at the station. This is the BCMP product-form case of
QD-AMVA (Casale, Perez, Wang, IFIP PERFORMANCE 2015):
D_{i,r}(n) = theta_{i,r} * beta_{i,r}(n_{i,r}). The handle returns a length-R
vector [beta_{i,1}(n), beta_{i,2}(n)], of which the solver picks class r.
Contrast ld_joint_dependence.py, where the scalar min(ni[0], c) reads a foreign
class marginal and is therefore non-product-form (set_joint_dependence).
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
N = 16  # number of jobs in class 1
c = 2   # number of servers
print('=== Load-Dependent Service - Class Dependence (product-form) ===\n')
cdmodel = Network('model')
delay = Delay(cdmodel, 'Delay')
queue = Queue(cdmodel, 'Queue1', SchedStrategy.PS)
job_class1 = ClosedClass(cdmodel, 'Class1', N, delay, 0)
job_class2 = ClosedClass(cdmodel, 'Class2', N // 2, delay, 0)
delay.set_service(job_class1, Exp.fit_mean(1.0))
delay.set_service(job_class2, Exp.fit_mean(2.0))
queue.set_service(job_class1, Exp.fit_mean(1.5))
queue.set_service(job_class2, Exp.fit_mean(2.5))
# beta_{i,r}(n_{i,r}): class 1 scales up to c servers with its OWN count, class
# 2 is always single-server. Both entries read only their own marginal, so the
# demands satisfy the product-form recurrence. Peak rate scaling [c, 1] per

    ... (11 source lines omitted; full implementation: python/examples/advanced/)

print(cd_avg_table_cd)
# JMT is not solved here: the JSIM writer has no representation for the
# class-dependence handle, so SolverJMT rejects the model rather than
# silently solving it unscaled (see SolverJMT.getFeatureSet).
print('\nNote: Class-dependent (product-form) service scales each class by its')
print('      own per-class population, preserving the BCMP product form.')
```

Running this edition prints

```
=== Load-Dependent Service - Class Dependence (product-form) ===

CTMC (exact):
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.88196  0.88196       1       1  0.88196  0.88196
Delay    Class2    0.27082  0.27082       2       2  0.13541  0.13541
Queue1   Class1     15.118  0.66147  17.141  17.141  0.88196  0.88196
Queue1   Class2     7.7292  0.33853  57.079  57.079  0.13541  0.13541

MVA with QD method:
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.88016  0.88016       1       1  0.88016  0.88016
Delay    Class2    0.27023  0.27023       2       2  0.13512  0.13512
Queue1   Class1      15.12  0.66012  17.178  17.178  0.88016  0.88016
Queue1   Class2     7.7298  0.33779  57.208  57.208  0.13512  0.13512

Note: Class-dependent (product-form) service scales each class by its
      own per-class population, preserving the BCMP product form.
```

The rows repeat for each of the 2 solvers the script runs (CTMC, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.4 `ld_fes_multiclass`

The multiclass flow-equivalent server.
The example is built and solved as follows.

```
Multiclass FES Aggregation with Norton's Theorem Verification
```

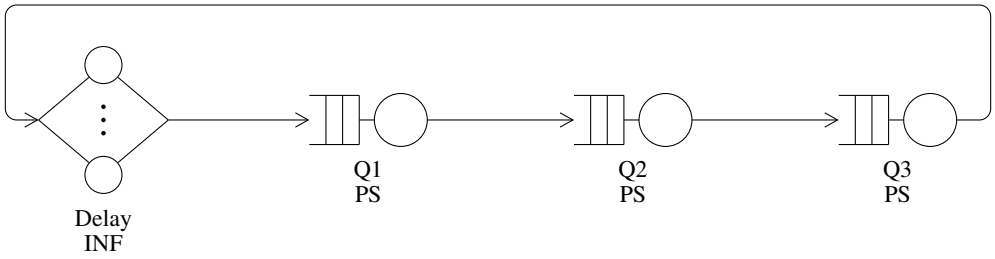

Figure 3.4: Topology of `ld_fes_multiclass`: a closed network over 4 queueing stations.

Nodes	4: Queue × 3, Delay
Classes	2: Class1 (closed, $N = 3$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Q1: PS; Q2: PS; Q3: PS
Service	Delay – Class1: Exp(1); Class2: Exp(0.6667) Q1 – Class1: Exp(2); Class2: Exp(1.25) Q2 – Class1: Exp(3.333); Class2: Exp(1.667) Q3 – Class1: Exp(2.5); Class2: Exp(1.429)
Solvers	MVA, NC

Table 3.4: Features of `ld_fes_multiclass`.

```
This example demonstrates Norton's theorem for closed multiclass
queueing networks: a subset of stations is replaced by a single
Flow-Equivalent Server (FES) whose class-dependent service rates equal the
throughputs of the isolated subnetwork.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
N1 = 3  # class-1 jobs
N2 = 2  # class-2 jobs
# Original 4-station tandem network with 2 classes
model = Network('OriginalModel')
delay = Delay(model, 'Delay')
q1 = Queue(model, 'Q1', SchedStrategy.PS)
q2 = Queue(model, 'Q2', SchedStrategy.PS)
q3 = Queue(model, 'Q3', SchedStrategy.PS)
jobclass1 = ClosedClass(model, 'Class1', N1, delay, 0)
jobclass2 = ClosedClass(model, 'Class2', N2, delay, 0)
delay.set_service(jobclass1, Exp.fit_mean(1.0))
delay.set_service(jobclass2, Exp.fit_mean(1.5))
q1.set_service(jobclass1, Exp.fit_mean(0.5))
q1.set_service(jobclass2, Exp.fit_mean(0.8))
q2.set_service(jobclass1, Exp.fit_mean(0.3))
q2.set_service(jobclass2, Exp.fit_mean(0.6))
q3.set_service(jobclass1, Exp.fit_mean(0.4))
q3.set_service(jobclass2, Exp.fit_mean(0.7))
P = model.init_routing_matrix()

    ... (8 source lines omitted; full implementation: python/examples/advanced/)

result = ModelAdapter.aggregate_fes(model, [q1, q2, q3])
fes_model = result['fes_model']
print('FES model created with %d stations' % fes_model.get_number_of_stations())
print('NC (FES model):')
avg_nc = NC(fes_model, method='exact').getAvgTable()
print(avg_nc)
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.89626  0.89626        1        1  0.89626  0.89626
Delay    Class2    0.54062  0.54062      1.5      1.5  0.36042  0.36042
Q1       Class1    0.97971  0.44813   1.0931   1.0931  0.89626  0.89626
Q1       Class2    0.63639  0.28833   1.7657   1.7657  0.36042  0.36042
Q2       Class1     0.4441  0.26888   0.4955   0.4955  0.89626  0.89626
Q2       Class2    0.34845  0.21625  0.96679  0.96679  0.36042  0.36042
```

```
Q3       Class1    0.67992   0.3585  0.75862  0.75862  0.89626  0.89626
Q3       Class2    0.47454  0.25229   1.3166   1.3166  0.36042  0.36042
FES model created with 2 stations
NC (FES model):
NC analysis [method: conv; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.89626  0.89626       1       1  0.89626  0.89626
Delay    Class2    0.54062  0.54062     1.5     1.5  0.36042  0.36042
FES      Class1     2.1037  0.51797  2.3472  2.3472  0.89626  0.89626
FES      Class2     1.4594  0.20829  4.0491  4.0491  0.36042  0.36042
```

The rows repeat for each of the 2 solvers the script runs (MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.5 `ld_fes_singleclass`

A flow-equivalent server replacing a subnetwork, single class.

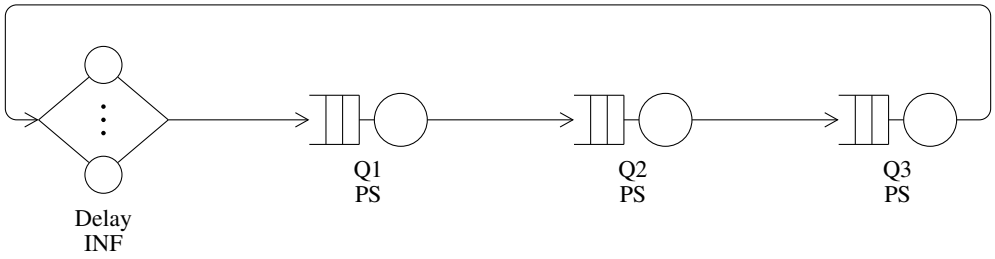

Figure 3.5: Topology of `ld_fes_singleclass`: a closed network over 4 queueing stations.

Nodes	4: Queue × 3, Delay
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Delay: INF; Q1: PS; Q2: PS; Q3: PS
Service	Delay – Class1: Exp(0.2) Q1 – Class1: Exp(0.6667) Q2 – Class1: Exp(1) Q3 – Class1: Exp(1.25)
Solvers	MVA, NC

Table 3.5: Features of `ld_fes_singleclass`.

The example is built and solved as follows.

```
Single-class FES Aggregation with Norton's Theorem Verification
This example demonstrates Norton's theorem for a closed single-class
queueing network: a subset of stations is replaced by a single
Flow-Equivalent Server (FES) whose load-dependent service rates
equal the throughputs of the isolated subnetwork.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
N1 = 5  # number of jobs
# Original 4-station tandem network with 1 class
model = Network('OriginalModel')
delay = Delay(model, 'Delay')
q1 = Queue(model, 'Q1', SchedStrategy.PS)
q2 = Queue(model, 'Q2', SchedStrategy.PS)
q3 = Queue(model, 'Q3', SchedStrategy.PS)
jobclass1 = ClosedClass(model, 'Class1', N1, delay, 0)
delay.set_service(jobclass1, Exp.fit_mean(5.0))
q1.set_service(jobclass1, Exp.fit_mean(1.5))
q2.set_service(jobclass1, Exp.fit_mean(1.0))
q3.set_service(jobclass1, Exp.fit_mean(0.8))
P = model.init_routing_matrix()
P[0][0] = model.serial_routing([delay, q1, q2, q3])
model.link(P)
# Solve original model
```

```
print('MVA (original):')
avg_orig = MVA(model, method='exact').getAvgTable()
print(avg_orig)
# Aggregate Q1, Q2, Q3 into FES and solve with NC (convolution)
result = ModelAdapter.aggregate_fes(model, [q1, q2, q3])
fes_model = result['fes_model']
print('FES model created with %d stations' % fes_model.get_number_of_stations())
print('NC (FES model):')
avg_nc = NC(fes_model, method='exact').getAvgTable()
print(avg_nc)
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
Delay    Class1     2.3333   2.3333        5       5  0.46666  0.46666
Q1       Class1     1.4069  0.69999   3.0148  3.0148  0.46666  0.46666
Q2       Class1    0.72959  0.46666   1.5634  1.5634  0.46666  0.46666
Q3       Class1    0.53021  0.37333   1.1362  1.1362  0.46666  0.46666
FES model created with 2 stations
NC (FES model):
NC analysis [method: conv; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
Delay    Class1     2.3333   2.3333        5       5  0.46666  0.46666
FES      Class1     2.6667   0.7705   5.7145  5.7145  0.46666  0.46666
```

The rows repeat for each of the 2 solvers the script runs (MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.6 ld_global_dependence

A globally state-dependent (Whittle) model: `setGlobalDependence` scales the service rate by a function of the FULL station-class population matrix rather than of the population local to one station. The scaling used here satisfies the Whittle balance property, so the chain is reversible, keeps a product form and is insensitive to the service distribution beyond its mean.

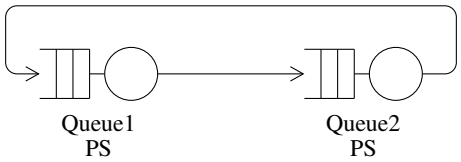

Figure 3.6: Topology of `ld_global_dependence`: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	1: Class1 (closed, $N = 3$)
Scheduling	Queue1: PS; Queue2: PS
Service	Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(2)
Solvers	CTMC, SSA

Table 3.6: Features of `ld_global_dependence`.

The example is built and solved as follows.

```
set_global_dependence declares a rate scaling phi(n) over the FULL
(nstations, nclasses) population matrix, not just the population local to one
station. Here two PS stations share one unit of capacity, phi_s(n) = n_s/|n|,
which is the single-link allocation every alpha-fair rule collapses to.
phi satisfies the Whittle balance property
    phi_s(n) phi_t(n-e_s) = phi_t(n) phi_s(n-e_t),
so the chain is reversible, has the product form pi(n) ~ Phi(n) prod rho**n and
is INSENSITIVE: the means below do not change when the exponential service is
```

```
replaced by an Erlang or a hyperexponential of the same mean.
MATLAB twin: matlab/examples/advanced/loadDependent/ld_global_dependence.m
                        SolverCTMC, SolverSSA)
N = 3  # number of jobs
model = Network('model')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
class1 = ClosedClass(model, 'Class1', N, queue1, 0)
queue1.setService(class1, Exp.fitMean(1.0))
queue2.setService(class1, Exp.fitMean(0.5))
model.link(Network.serialRouting(queue1, queue2))
sn = model.getStruct()
# getNodeIndex is 1-based while nodeToStation is a 0-based array
i1 = int(sn.nodeToStation[model.getNodeIndex('Queue1') - 1])
i2 = int(sn.nodeToStation[model.getNodeIndex('Queue2') - 1])
def share(n):
    """phi as SolverCTMC sees it: an (nstations, nclasses) matrix of scalings."""
    v = np.ones(n.shape)

    ... (13 source lines omitted; full implementation: python/examples/advanced/)

# one station's population slice), so the model is routed to the serial engine.
print(SolverSSA(model, seed=23000, samples=200000).getAvgTable())
# The balance property is checkable, and is what separates a Whittle network from
# an arbitrary state-dependent rate.
viol, _ = sn_gd_balance(lambda n: np.asarray(n) / max(np.sum(n), 1e-300), [4, 4])
print('\nworst relative balance violation: %.2e (balanced when ~0)' % viol)
```

Running this edition prints

```
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: sync; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass  QLen     Util  RespT  ResidT     ArvR     Tput
Queue1   Class1       2  0.66667      3       3  0.66667  0.66667
Queue2   Class1       1  0.33333    1.5     1.5  0.66667  0.66667
SSA analysis [method: default/serial; type: approximate, randomized; lang: python; env: python] completed
    in <t>s.
Station  JobClass     QLen     Util  RespT  ResidT     ArvR     Tput
Queue1   Class1     2.0021  0.66524      3       3  0.66524  0.66738
Queue2   Class1    0.99787  0.33369    1.5     1.5  0.66738  0.66524

worst relative balance violation: 1.85e-16 (balanced when ~0)
```

The rows repeat for each of the 2 solvers the script runs (CTMC, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.7 `ld_joint_dependence`

Joint dependence on the populations of several classes.

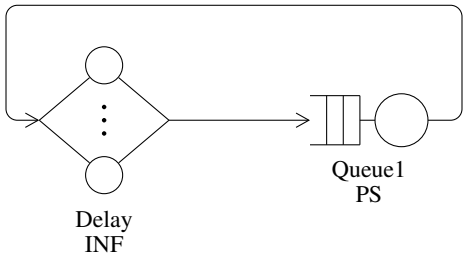

Figure 3.7: Topology of `ld_joint_dependence`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
Load-Dependent Service - Joint Dependence
This example demonstrates joint-dependent (NON-product-form) service rates: the
service rate reads a foreign class marginal (class-1 population), so it is
```

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 16$), Class2 (closed, $N = 8$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(0.5) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.4)
Solvers	CTMC, MVA

Table 3.7: Features of `ld_joint_dependence`.

```
not expressible as a product-form beta_{i,r}(n_{i,r}). In this example,
multi-server behavior is applied only for class 1 jobs.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
N = 16  # number of jobs in class 1
c = 2   # number of servers
print('=== Load-Dependent Service - Joint Dependence ===\n')
# Joint-dependent (non-product-form) model
jdmodel = Network('model')
delay = Delay(jdmodel, 'Delay')
queue = Queue(jdmodel, 'Queue1', SchedStrategy.PS)
job_class1 = ClosedClass(jdmodel, 'Class1', N, delay, 0)
job_class2 = ClosedClass(jdmodel, 'Class2', N // 2, delay, 0)
delay.set_service(job_class1, Exp.fit_mean(1.0))
delay.set_service(job_class2, Exp.fit_mean(2.0))
queue.set_service(job_class1, Exp.fit_mean(1.5))
queue.set_service(job_class2, Exp.fit_mean(2.5))
# Joint dependence (non-product-form): rate reads the class-1 marginal only,
# shared across classes. ni is a vector where ni[r] is the class-r population.
queue.set_joint_dependence(lambda ni: min(ni[0], c), c)  # peak rate scaling = c (Util = T*S/
    c)
P = jdmodel.init_routing_matrix()
P[0][0] = jdmodel.serial_routing([delay, queue])
P[1][1] = jdmodel.serial_routing([delay, queue])

    ... (8 source lines omitted; full implementation: python/examples/advanced/)

# joint-dependence handle, so SolverJMT rejects the model rather than
# silently solving it unscaled (see SolverJMT.getFeatureSet).
print('\nNote: Joint-dependent service is non-product-form (see set_joint_dependence).')
print('      In this example, service rate scales with Class 1 population only,')
print('      modeling c servers available exclusively for Class 1 jobs.')
print('      Class 2 jobs do not benefit from multi-server parallelism.')
```

Running this edition prints

```
=== Load-Dependent Service - Joint Dependence ===

CTMC (exact):
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.89231  0.89231       1       1  0.89231  0.89231
Delay    Class2    0.52923  0.52923       2       2  0.26461  0.26461
Queue1   Class1     15.108  0.66923  16.931  16.931  0.89231  0.89231
Queue1   Class2     7.4708  0.33077  28.233  28.233  0.26461  0.26461

MVA with QD method:
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.88996  0.88996       1       1  0.88996  0.88996
Delay    Class2    0.52788  0.52788       2       2  0.26394  0.26394
Queue1   Class1      15.11  0.66747  16.978  16.978  0.88996  0.88996
Queue1   Class2     7.4721  0.32992   28.31   28.31  0.26394  0.26394

Note: Joint-dependent service is non-product-form (see set_joint_dependence).
      In this example, service rate scales with Class 1 population only,
      modeling c servers available exclusively for Class 1 jobs.
      Class 2 jobs do not benefit from multi-server parallelism.
```

The rows repeat for each of the 2 solvers the script runs (CTMC, MVA), which is the point of the

example: the same model read by methods of different cost and different exactness.

3.1.8 `ld_multiserver_fcfs`

The FCFS counterpart, with class-dependent scaling on top.

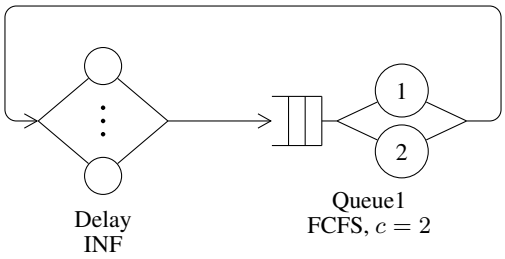

Figure 3.8: Topology of `ld_multiserver_fcfs`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 16$)
Scheduling	Delay: INF; Queue1: FCFS ($c = 2$)
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(0.6667)
Solvers	NC, CTMC, MVA, JMT

Table 3.8: Features of `ld_multiserver_fcfs`.

The example is built and solved as follows.

```
Load-Dependent Service - Multi-Server FCFS Queue
This example demonstrates load-dependent service for modeling multi-server
FCFS queues, comparing three approaches: standard multi-server, load-dependent,
and class-dependent service rates.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
N = 16  # number of jobs
c = 2   # number of servers
print('=== Load-Dependent Service - Multi-Server FCFS Queue ===\n')
# Standard multi-server model
print('--- Standard Multi-Server Model ---')
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
job_class = ClosedClass(model, 'Class1', N, delay, 0)
delay.set_service(job_class, Exp.fit_mean(1.0))
queue.set_service(job_class, Exp.fit_mean(1.5))
queue.set_number_of_servers(c)
model.link(model.serial_routing([delay, queue]))
ms_t = NC(model).getAvgTable()
print('NC Solver:')
print(ms_t)
# Load-dependent model
print('\n--- Load-Dependent Model ---')
ldmodel = Network('model')
delay = Delay(ldmodel, 'Delay')

    ... (47 source lines omitted; full implementation: python/examples/advanced/)

cd_avg_table_cd = MVA(cdmodel, method='qd').getAvgTable()
print(cd_avg_table_cd)
# JMT is not solved here: the JSIM writer has no representation for the
# class-dependence handle, so SolverJMT rejects the model rather than
# silently solving it unscaled (see SolverJMT.getFeatureSet). The lld
# model above is JMT-solvable because lldscaling maps onto the server count.
```

Running this edition prints

```
=== Load-Dependent Service - Multi-Server FCFS Queue ===

--- Standard Multi-Server Model ---
NC analysis [method: default/exact/gld; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
NC Solver:
Station  JobClass    QLen    Util  RespT  ResidT    ArvR    Tput
Delay    Class1    1.3333  1.3333      1       1  1.3333  1.3333
Queue1   Class1    14.667       1     11      11  1.3333  1.3333

--- Load-Dependent Model ---

CTMC (exact):
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
Station  JobClass    QLen    Util  RespT  ResidT    ArvR    Tput
Delay    Class1    1.3333  1.3333      1       1  1.3333  1.3333

... (52 captured-output lines omitted; rerun the source example for the full output)

Delay    Class1    1.3333  1.3333      1       1  1.3333  1.3333
Queue1   Class1    14.667       1     11      11  1.3333  1.3333

MVA with QD method:
MVA analysis [method: qd; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util   RespT  ResidT    ArvR    Tput
Delay    Class1    1.3258   1.3258       1       1  1.3258  1.3258
Queue1   Class1    14.674  0.99438  11.068  11.068  1.3258  1.3258
```

The rows repeat for each of the 4 solvers the script runs (NC, CTMC, MVA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.9 `ld_multiserver_ps`

A multiserver PS station written as a load-dependent station, and the rate vector that reproduces it.

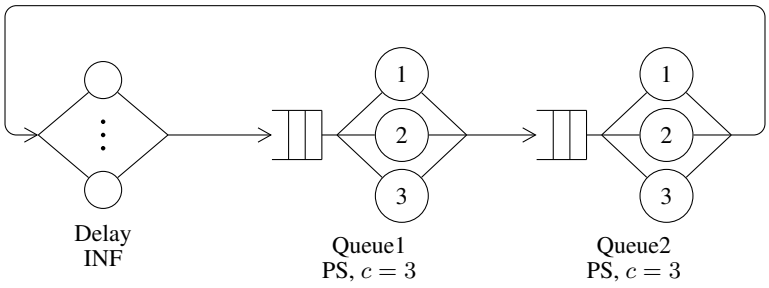

Figure 3.9: Topology of `ld_multiserver_ps`: a closed network over 3 queueing stations.

Nodes	3: Queue $\times$ 2, Delay
Classes	2: Class1 (closed, $N = 4$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS ($c = 3$); Queue2: PS ($c = 3$)
Service	Delay – Class1: Exp(1); Class2: Exp(0.5) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.4) Queue2 – Class1: Exp(0.2857); Class2: Exp(0.2222)
Solvers	MVA, CTMC, NC, JMT

Table 3.9: Features of `ld_multiserver_ps`.

The example is built and solved as follows.

```
Load-Dependent Service - Multi-Server PS Queue
This example demonstrates how to model a multi-server processor-sharing queue
using load-dependent service rates. Compares standard multi-server model with
```

```
load-dependent approximation using multiple solvers.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
N = 4  # number of jobs
c = 3  # number of servers
print('=== Load-Dependent Service - Multi-Server PS Queue ===\n')
# Standard multi-server model
print('--- Standard Multi-Server Model ---')
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
job_class1 = ClosedClass(model, 'Class1', N, delay, 0)
job_class2 = ClosedClass(model, 'Class2', N // 2, delay, 0)
delay.set_service(job_class1, Exp.fit_mean(1.0))
delay.set_service(job_class2, Exp.fit_mean(2.0))
queue1.set_service(job_class1, Exp.fit_mean(1.5))
queue1.set_service(job_class2, Exp.fit_mean(2.5))
queue1.set_number_of_servers(c)
queue2.set_service(job_class1, Exp.fit_mean(3.5))
queue2.set_service(job_class2, Exp.fit_mean(4.5))
queue2.set_number_of_servers(c)
P = model.init_routing_matrix()

    ... (48 source lines omitted; full implementation: python/examples/advanced/)

print(lld_avg_table_qd)
print('\nJMT Simulation:')
lld_avg_table_jmt = JMT(ldmodel, seed=23000).getAvgTable()
print(lld_avg_table_jmt)
print('\nNote: Load dependence allows modeling multi-server queues by scaling')
print('      service rates based on the number of jobs present.')
```

Running this edition prints

```
=== Load-Dependent Service - Multi-Server PS Queue ===

--- Standard Multi-Server Model ---
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
MVA (exact):
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.54674  0.54674       1       1  0.54674  0.54674
Delay    Class2    0.36944  0.36944       2       2  0.18472  0.18472
Queue1   Class1    0.85639  0.27337  1.5664  1.5664  0.54674  0.54674
Queue1   Class2    0.48077  0.15394  2.6027  2.6027  0.18472  0.18472
Queue2   Class1     2.5969  0.63786  4.7498  4.7498  0.54674  0.54674
Queue2   Class2     1.1498  0.27708  6.2244  6.2244  0.18472  0.18472

--- Load-Dependent Model ---

CTMC (exact):

... (76 captured-output lines omitted; rerun the source example for the full output)

1   Delay   Class2  0.37807  0.37807  1.99278  1.99278  0.18399  0.18418
2  Queue1   Class1  0.84282  0.26770  1.53077  1.53077  0.55206  0.55836
3  Queue1   Class2  0.47861  0.14715  2.57155  2.57155  0.18418  0.18434
4  Queue2   Class1  2.60813  0.64247  4.74617  4.74617  0.55836  0.55871
5  Queue2   Class2  1.13935  0.27307  6.15718  6.15718  0.18434  0.18397

Note: Load dependence allows modeling multi-server queues by scaling
      service rates based on the number of jobs present.
```

The rows repeat for each of the 4 solvers the script runs (MVA, CTMC, NC, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.10 `ld_multiserver_ps_twoclasses`

The two-class version.
The example is built and solved as follows.

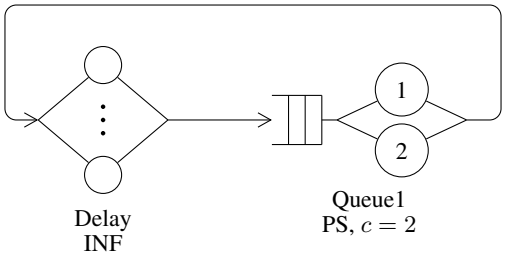

Figure 3.10: Topology of `ld_multiserver_ps_twoclasses`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 4$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(0.5) Queue1 – Class1: Exp(0.6667); Class2: Exp(0.4)
Solvers	MVA, CTMC, NC, JMT

Table 3.10: Features of `ld_multiserver_ps_twoclasses`.

```
Load-Dependent Service - Multi-Server PS Queue with Two Classes
This example demonstrates load-dependent service rates for a multi-server
processor-sharing queue with two job classes.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
N = 4  # number of jobs in class 1
c = 2  # number of servers
print('=== Load-Dependent Service - Multi-Server PS Queue (Two Classes) ===\n')
# Standard multi-server model
print('--- Standard Multi-Server Model ---')
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.PS)
job_class1 = ClosedClass(model, 'Class1', N, delay, 0)
job_class2 = ClosedClass(model, 'Class2', N // 2, delay, 0)
delay.set_service(job_class1, Exp.fit_mean(1.0))
delay.set_service(job_class2, Exp.fit_mean(2.0))
queue.set_service(job_class1, Exp.fit_mean(1.5))
queue.set_service(job_class2, Exp.fit_mean(2.5))
queue.set_number_of_servers(c)
P = model.init_routing_matrix()
P[0][0] = model.serial_routing([delay, queue])
P[1][1] = model.serial_routing([delay, queue])
model.link(P)
ms_t = MVA(model, 'exact').getAvgTable()
print('MVA (exact):')

    ... (38 source lines omitted; full implementation: python/examples/advanced/)

print(lld_avg_table_qd)
print('\nJMT Simulation:')
lld_avg_table_jmt = JMT(ldmodel, seed=23000, samples=5000).getAvgTable()
print(lld_avg_table_jmt)
print('\nNote: Load dependence models multi-server behavior across multiple classes.')
print('      Service rate scales with total queue length up to c servers.')
```

Running this edition prints

```
=== Load-Dependent Service - Multi-Server PS Queue (Two Classes) ===

--- Standard Multi-Server Model ---
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
MVA (exact):
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1     0.8975   0.8975       1       1   0.8975   0.8975
Delay    Class2    0.51157  0.51157       2       2  0.25579  0.25579
```

```
Queue1   Class1     3.1025  0.67313  3.4568  3.4568   0.8975   0.8975
Queue1   Class2     1.4884  0.31973   5.819   5.819  0.25579  0.25579

--- Load-Dependent Model ---

CTMC (exact):
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

... (58 captured-output lines omitted; rerun the source example for the full output)

  Station JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0   Delay   Class1  0.92463  0.92463  0.98552  0.98552  0.92113  0.91876
1   Delay   Class2  0.49666  0.49666  1.92848  1.92848  0.26303  0.26009
2  Queue1   Class1  3.07537  0.67693  3.38339  3.38339  0.91876  0.91533
3  Queue1   Class2  1.50334  0.31834  5.73774  5.73774  0.26009  0.26312

Note: Load dependence models multi-server behavior across multiple classes.
      Service rate scales with total queue length up to c servers.
```

The rows repeat for each of the 4 solvers the script runs (MVA, CTMC, NC, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.1.11 `ld_whittle_bandwidth`

An open bandwidth-sharing network in which one route holds several links at once, which no per-station rate scaling can express. Capacity is shared by balanced fairness, whose rate function satisfies the Whittle property by construction, so the stationary law is again of product form and insensitive.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
One route holds SEVERAL links at once, which no per-station rate scaling can
express. This is the 2-link linear network: route 1 crosses both links, routes 2
and 3 use one link each.
    A = [1 1 0;      link 1 carries routes 1 and 2
         1 0 1]      link 2 carries routes 1 and 3
Capacity is shared by BALANCED FAIRNESS, whose rates come from the recursion
    Phi(n) = max_l (1/C_l) sum_{s in l} Phi(n-e_s),  x_s(n) = Phi(n-e_s)/Phi(n).
That construction satisfies the Whittle balance property by design, so the
stationary law is pi(n) ~ Phi(n) prod rho_s**n_s and is insensitive.
Each route is modelled as its own PS queue fed by its own open class, so the
queue populations ARE the coordinates of the Whittle state n.
MATLAB twin: matlab/examples/advanced/loadDependent/ld_whittle_bandwidth.m
                         SchedStrategy, Sink, SolverCTMC, Source)
A = np.array([[1, 1, 0], [1, 0, 1]])
C = np.array([1.0, 1.0])
nu = np.array([0.20, 0.30, 0.30])  # flow arrival rates
mu = np.array([1.00, 1.00, 1.00])  # 1/mean file size
cutoff = 6                         # per-route truncation of the open state space
def balance_function(A, C, cut):
    """Phi(n) = max_l (1/C_l) sum_{s in l} Phi(n-e_s), over the bounded lattice."""
    S = A.shape[1]
    base = np.asarray(cut, dtype=int) + 1
    states = np.array(list(itertools.product(*[range(b) for b in base])), dtype=int)
    # itertools.product varies the LAST coordinate fastest; index accordingly
    index = {tuple(n): k for k, n in enumerate(states)}
    Phi = np.zeros(len(states))

    ... (66 source lines omitted; full implementation: python/examples/advanced/)

print(SolverCTMC(model, cutoff=cutoff).getAvgTable())
# Cross-check against the closed-form product form. Truncating a reversible chain
# preserves the conditional law, so this agrees to solver precision rather than to
# a truncation-limited tolerance.
En = bf_means(Phi, index, nu / mu)
```

```
print('\nproduct form E[n] = [%.6f %.6f %.6f]' % tuple(En))
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.2 Finite capacity regions

A finite capacity region caps the number of jobs inside a set of stations. A job that would breach the cap is either dropped or made to wait outside the region, per class, and per-class caps may be declared alongside the global one. The region is a property of the model, not of a station, which is what lets it span several stations.

The flagship comparison is `fcr_mm1kdrop`: a region of capacity K around a single queue, with dropping, is an M/M/1/K queue, and the script builds both models and solves them side by side.

```
# Model 1: one queue inside a region of capacity K, with dropping
model1 = Network('FCR Dropping')
source1 = Source(model1, 'Source')
queue1 = Queue(model1, 'Queue', SchedStrategy.FCFS)
sink1 = Sink(model1, 'Sink')
jobclass1 = OpenClass(model1, 'Class1', 0)
source1.set_arrival(jobclass1, Exp.fit_rate(0.8))
queue1.set_service(jobclass1, Exp.fit_rate(1.0))
model1.link(Network.serial_routing([source1, queue1, sink1]))

fcr = model1.add_region(queue1)
fcr.set_global_max_jobs(K)
fcr.set_drop_rule(jobclass1, True)    # True = drop, False = wait outside
```

3.2.1 `fcr_constraints`

Per-class capacities inside one region, and the interaction with the global limit.

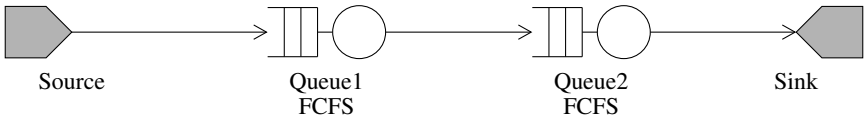

Figure 3.11: Topology of `fcr_constraints`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	2: HighPriority (open), LowPriority (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – HighPriority: Exp(0.3); LowPriority: Exp(0.5)
Service	Queue1 – HighPriority: Exp(1); LowPriority: Exp(0.8) Queue2 – HighPriority: Exp(1.2); LowPriority: Exp(1)
Solvers	JMT

Table 3.11: Features of `fcr_constraints`.

The example is built and solved as follows.

```
Finite Capacity Region - Constraint Types
This example demonstrates different FCR constraint types:
- Global max jobs: limits total jobs across all classes
- Per-class max jobs: limits jobs of a specific class
- Drop rule: determines blocking vs dropping behavior
# Create a multiclass open network
model = Network('FCR Constraints Demo')
```

```
# Nodes
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Job classes
high_priority = OpenClass(model, 'HighPriority', 0)
low_priority = OpenClass(model, 'LowPriority', 1)
# Arrival and service rates
source.set_arrival(high_priority, Exp.fit_rate(0.3))  # High priority: lower arrival rate
source.set_arrival(low_priority, Exp.fit_rate(0.5))   # Low priority: higher arrival rate
queue1.set_service(high_priority, Exp.fit_rate(1.0))
queue1.set_service(low_priority, Exp.fit_rate(0.8))
queue2.set_service(high_priority, Exp.fit_rate(1.2))
queue2.set_service(low_priority, Exp.fit_rate(1.0))
# Routing: both classes go through both queues
P = model.init_routing_matrix()
P.set(high_priority, high_priority, source, queue1, 1.0)
P.set(high_priority, high_priority, queue1, queue2, 1.0)

    ... (15 source lines omitted; full implementation: python/examples/advanced/)

fcr.set_drop_rule(high_priority, True)    # True = drop
fcr.set_drop_rule(low_priority, True)     # True = drop
# Solve with JMT
solver = JMT(model, seed=23000, samples=100000)
avg_table = solver.get_avg_table()
print(avg_table)
```

Running this edition prints

```
Warning [refresh_struct]: Priority classes are specified but no priority-aware scheduling policy (PSPRIO,
    DPSPRIO, GPSPRIO, HOL, FCFSPRIO, FCFSPRPRIO, FCFSPIPRIO, LCFSPRIO, LCFSPRPRIO, LCFSPIPRIO, SRPTPRIO)
    is used in the model. Priorities will be ignored.
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass         QLen     Util    RespT  ResidT     ArvR     Tput
Source   HighPriority        0        0        0       0        0  0.30214
Source   LowPriority         0        0        0       0        0   0.4979
Queue1   HighPriority  0.23559   0.1681   1.4066  1.4066   0.1674   0.1674
Queue1   LowPriority   0.46107  0.35021   1.6579  1.6579  0.27867  0.27969
Queue2   HighPriority   0.1866  0.14182   1.1209  1.1209   0.1674  0.16751
Queue2   LowPriority    0.3488  0.27839   1.2433  1.2433  0.27969  0.27998
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by JMT.

3.2.2 fcr_lincon

A linear constraint over the class populations, the general form of a region limit.

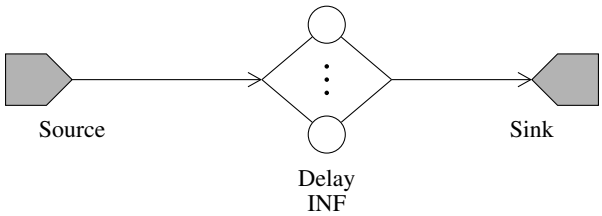

Figure 3.12: Topology of fcr_lincon: an open network over 1 queueing station.

The example is built and solved as follows.

```
Linear Admission Constraints for Finite Capacity Regions
Open network with a Delay node inside an FCR with DROP policy and
```

Nodes	3: Source, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF
Arrivals	Source – Class1: Exp(0.3); Class2: Exp(0.2)
Service	Delay – Class1: Exp(1); Class2: Exp(0.8)
Solvers	LDES

Table 3.12: Features of `fcr_lincon`.

```
general linear admission constraints: A*n <= b.
Constraint: 2*n_1 + 1*n_2 <= 5, 1*n_1 + 3*n_2 <= 7
(general weighted cross-class coupling)
model = Network('FCR LinCon')
source = Source(model, 'Source')
delay = Delay(model, 'Delay')
sink = Sink(model, 'Sink')
class1 = OpenClass(model, 'Class1', 0)
class2 = OpenClass(model, 'Class2', 0)
source.set_arrival(class1, Exp(0.3))
source.set_arrival(class2, Exp(0.2))
delay.set_service(class1, Exp(1.0))
delay.set_service(class2, Exp(0.8))
P = model.init_routing_matrix()
P.set(class1, class1, source, delay, 1.0)
P.set(class1, class1, delay, sink, 1.0)
P.set(class2, class2, source, delay, 1.0)
P.set(class2, class2, delay, sink, 1.0)
model.link(P)
# Add FCR with DROP policy and linear constraints
fcr = model.add_region(delay)
fcr.set_class_max_jobs(class1, 1000)
fcr.set_class_max_jobs(class2, 1000)
fcr.set_drop_rule(class1, DropStrategy.DROP)

    ... (3 source lines omitted; full implementation: python/examples/advanced/)

    np.array([[2.0, 1.0], [1.0, 3.0]]),
    np.array([5.0, 7.0])
)
solver = LDES(model, seed=23000, samples=500000)
avg_table = solver.avg_table()
print(avg_table)
```

Running this edition prints

```
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 500000 --seed 23000
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Source   Class1          0        0       0       0        0      0.3
Source   Class2          0        0       0       0        0      0.2
Delay    Class1    0.28914  0.28869  1.0012  1.0012  0.29193  0.28869
Delay    Class2    0.24296  0.24202  1.2547  1.2547  0.19575  0.19362
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `LDES`.

3.2.3 `fcr_lossn`

A loss network: every request that cannot be admitted is lost, and the blocking probabilities are the quantity of interest.
The example is built and solved as follows.

```
Loss Network with Finite Capacity Region (NC Solver)
This example demonstrates the NC solver's ability to analyze open
loss networks using the Erlang fixed-point approximation.
The model has a multiclass Delay node inside an FCR with DROP policy.
```

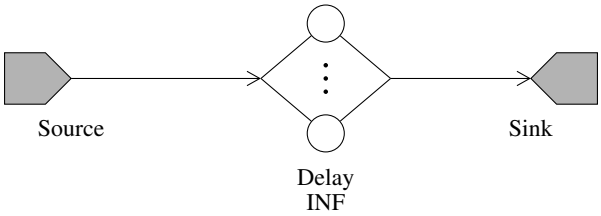

Figure 3.13: Topology of `fcr_lossn`: an open network over 1 queueing station.

Nodes	3: Source, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF
Arrivals	Source – Class1: Exp(0.3); Class2: Exp(0.2)
Service	Delay – Class1: Exp(1); Class2: Exp(0.8)
Solvers	JMT

Table 3.13: Features of `fcr_lossn`.

```
# Create network
model = Network('FCR Loss Network')
# Add nodes
source = Source(model, 'Source')
delay = Delay(model, 'Delay')
sink = Sink(model, 'Sink')
# Add job classes
class1 = OpenClass(model, 'Class1', 0)
class2 = OpenClass(model, 'Class2', 1)
# Set arrival and service rates
lambda1 = 0.3
lambda2 = 0.2
mu1 = 1.0
mu2 = 0.8
source.set_arrival(class1, Exp.fit_rate(lambda1))
source.set_arrival(class2, Exp.fit_rate(lambda2))
delay.set_service(class1, Exp.fit_rate(mu1))
delay.set_service(class2, Exp.fit_rate(mu2))
# Create routing matrix
P = model.init_routing_matrix()
P.set(class1, class1, source, delay, 1.0)
P.set(class1, class1, delay, sink, 1.0)

    ... (17 source lines omitted; full implementation: python/examples/advanced/)

# Run JMT for comparison
print('\nRunning JMT for comparison...')
solver_jmt = JMT(model, seed=23000, samples=500000)
avg_table_jmt = solver_jmt.get_avg_table()
print('\nJMT Results:')
print(avg_table_jmt)
```

Running this edition prints

```
Running NC solver (lossn method)...
Warning [refresh_struct]: Priority classes are specified but no priority-aware scheduling policy (PSPRIO,
     DPSPRIO, GPSPRIO, HOL, FCFSPRIO, FCFSPRPRIO, FCFSPIPRIO, LCFSPRIO, LCFSPRPRIO, LCFSPIPRIO, SRPTPRIO)
     is used in the model. Priorities will be ignored.
Warning [SolverFeatureSet]: Some features are not supported by the chosen solver (feature: Region).

NC Results:
Station  JobClass     QLen     Util  RespT  ResidT     ArvR     Tput
Source   Class1          0        0      0       0        0  0.29898
Source   Class2          0        0      0       0        0  0.19958
Delay    Class1    0.29898  0.29898      1       1  0.29898  0.29898
Delay    Class2    0.24947  0.24947   1.25    1.25  0.19958  0.19958

Running JMT for comparison...
```

```
Warning [refresh_struct]: Priority classes are specified but no priority-aware scheduling policy (PSPRIO,
     DPSPRIO, GPSPRIO, HOL, FCFSPRIO, FCFSPRPRIO, FCFSPIPRIO, LCFSPRIO, LCFSPRPRIO, LCFSPIPRIO, SRPTPRIO)
     is used in the model. Priorities will be ignored.
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.

JMT Results:
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Source   Class1          0        0       0       0        0  0.29989
Source   Class2          0        0       0       0        0  0.20024
Delay    Class1    0.29949  0.29949  1.0024  1.0024  0.29913  0.29901
Delay    Class2    0.25002  0.25002  1.2487  1.2487  0.19982  0.19982
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by JMT.

3.2.4 fcr_mm1kdrop

A region of capacity K around one queue with dropping, which reproduces an M/M/1/K queue.

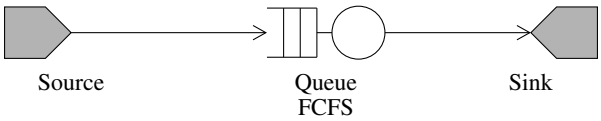

Figure 3.14: Topology of fcr_mm1kdrop: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(0.8)
Service	Queue – Class1: Exp(1)
Solvers	JMT

Table 3.14: Features of fcr_mm1kdrop.

The example is built and solved as follows.

```
Finite Capacity Region with Dropping vs M/M/1/K
This example shows that an FCR with dropping around a single queue
behaves like an M/M/1/K queue, where K is the FCR capacity.
Jobs arriving when the region is full are dropped (lost).
# Parameters
arrival_rate = 0.8
service_rate = 1.0
K = 3
# Model 1: Queue with FCR (dropping)
model1 = Network('FCR Dropping')
source1 = Source(model1, 'Source')
queue1 = Queue(model1, 'Queue', SchedStrategy.FCFS)
sink1 = Sink(model1, 'Sink')
jobclass1 = OpenClass(model1, 'Class1', 0)
source1.set_arrival(jobclass1, Exp.fit_rate(arrival_rate))
queue1.set_service(jobclass1, Exp.fit_rate(service_rate))
P1 = model1.init_routing_matrix()
P1.set(jobclass1, jobclass1, source1, queue1, 1.0)
P1.set(jobclass1, jobclass1, queue1, sink1, 1.0)
model1.link(P1)
# Add FCR with dropping - jobs are lost when region is full
fcr = model1.add_region(queue1)
fcr.set_global_max_jobs(K)
fcr.set_drop_rule(jobclass1, True)  # true = drop jobs
# Model 2: M/M/1/K using queue capacity
```

```
model2 = Network('M/M/1/K')

    ... (13 source lines omitted; full implementation: python/examples/advanced/)

solver1 = JMT(model1, seed=23000, samples=100000)
avgTable1 = solver1.get_avg_table()
print(avgTable1)
solver2 = JMT(model2, seed=23000, samples=100000)
avgTable2 = solver2.get_avg_table()
print(avgTable2)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util    RespT  ResidT    ArvR     Tput
Source   Class1         0        0        0       0       0  0.79526
Queue    Class1    1.2175  0.66118   1.8344  1.8344  0.6621  0.66208
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Source   Class1         0        0        0       0        0  0.79526
Queue    Class1    1.2212  0.66118   1.8344  1.8344  0.79526  0.66208
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.2.5 `fcr_mm1waitq`

The same region with waiting instead of dropping, so blocked jobs queue outside the region.

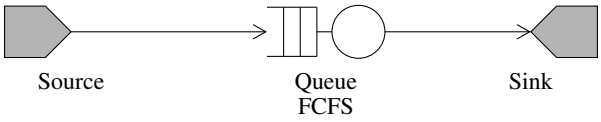

Figure 3.15: Topology of `fcr_mm1waitq`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(0.5)
Service	Queue – Class1: Exp(1)
Solvers	JMT

Table 3.15: Features of `fcr_mm1waitq`.

The example is built and solved as follows.

```
Finite Capacity Region with Blocking vs M/M/1
This example shows that an FCR with blocking (waitq) around a single
queue behaves identically to a standard M/M/1 queue, since jobs
simply wait when the region is "full" (no actual capacity limit effect).
# Parameters
arrival_rate = 0.5
service_rate = 1.0
# Model 1: Queue with FCR (blocking)
model1 = Network('FCR Blocking')
source1 = Source(model1, 'Source')
queue1 = Queue(model1, 'Queue', SchedStrategy.FCFS)
sink1 = Sink(model1, 'Sink')
```

```
jobclass1 = OpenClass(model1, 'Class1', 0)
source1.set_arrival(jobclass1, Exp.fit_rate(arrival_rate))
queue1.set_service(jobclass1, Exp.fit_rate(service_rate))
P1 = model1.init_routing_matrix()
P1.set(jobclass1, jobclass1, source1, queue1, 1.0)
P1.set(jobclass1, jobclass1, queue1, sink1, 1.0)
model1.link(P1)
# Add FCR with blocking - jobs wait when region is full
fcr = model1.add_region(queue1)
fcr.set_global_max_jobs(10)
fcr.set_drop_rule(jobclass1, False)  # False = block (wait)
# Model 2: Standard M/M/1 (no FCR)
model2 = Network('M/M/1')
source2 = Source(model2, 'Source')

    ... (10 source lines omitted; full implementation: python/examples/advanced/)

solver1 = JMT(model1, seed=23000, samples=100000)
avgTable1 = solver1.get_avg_node_table()
print(avgTable1)
solver2 = JMT(model2, seed=23000, samples=100000)
avgTable2 = solver2.get_avg_node_table()
print(avgTable2)
```

Running this edition prints

```
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
     Node JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  Source   Class1  0.00000  0.00000  0.00000  0.00000  0.00000  0.49879
1   Queue   Class1  0.97836  0.49392  1.96409  1.96409  0.49879  0.49923
2    Sink   Class1  0.00000  0.00000  0.00000  0.00000  0.49879  0.00000
3    FCR1   Class1  0.97860      NaN  1.96430  1.96432      NaN  0.49879
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
     Node JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
0  Source   Class1  0.0000  0.00000  0.0000  0.0000  0.00000  0.49879
1   Queue   Class1  0.9786  0.49392  1.9643  1.9643  0.49879  0.49923
2    Sink   Class1  0.0000  0.00000  0.0000  0.0000  0.49879  0.00000
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.2.6 `fcr_oqndrop`

A region spanning several stations of an open network, with dropping.

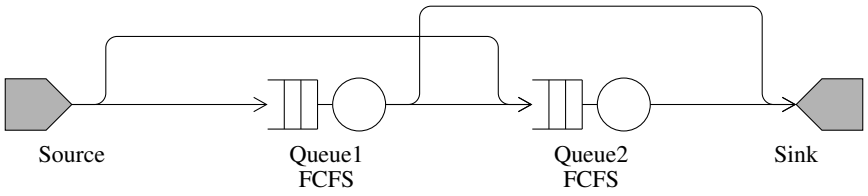

Figure 3.16: Topology of `fcr_oqndrop`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
Finite Capacity Region with Dropping (Multiclass Open Network)
This example creates a multiclass open network with 2 queues and a
finite capacity region with global and per-class constraints.
When the region is full, arriving jobs are dropped (lost).
# Create network
model = Network('FCR Dropping Example')
```

Nodes	4: Source, Queue × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.4); Class2: Exp(0.3)
Service	Queue1 – Class1: Exp(1); Class2: Exp(0.9) Queue2 – Class1: Exp(1.1); Class2: Exp(1)
Solvers	JMT

Table 3.16: Features of `fcr_oqndrop`.

```
# Add nodes
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Add job classes
class1 = OpenClass(model, 'Class1', 0)
class2 = OpenClass(model, 'Class2', 1)
# Set arrival and service rates
source.set_arrival(class1, Exp.fit_rate(0.4))
source.set_arrival(class2, Exp.fit_rate(0.3))
queue1.set_service(class1, Exp.fit_rate(1.0))
queue1.set_service(class2, Exp.fit_rate(0.9))
queue2.set_service(class1, Exp.fit_rate(1.1))
queue2.set_service(class2, Exp.fit_rate(1.0))
# Create routing matrix
P = model.init_routing_matrix()
# Class1 routing
P.set(class1, class1, source, queue1, 0.5)
P.set(class1, class1, source, queue2, 0.5)

    ... (17 source lines omitted; full implementation: python/examples/advanced/)

fcr.set_drop_rule(class2, True)     # True = drop jobs
# Run JMT
solver = JMT(model, seed=23000, samples=50000)
# Get results
avg_table = solver.get_avg_table()
print(avg_table)
```

Running this edition prints

```
Warning [refresh_struct]: Priority classes are specified but no priority-aware scheduling policy (PSPRIO,
     DPSPRIO, GPSPRIO, HOL, FCFSPRIO, FCFSPRPRIO, FCFSPIPRIO, LCFSPRIO, LCFSPRPRIO, LCFSPIPRIO, SRPTPRIO)
     is used in the model. Priorities will be ignored.
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Source   Class1         0        0       0        0        0  0.40034
Source   Class2         0        0       0        0        0  0.29982
Queue1   Class1    0.27805  0.18679  1.4724  0.73622   0.1885   0.1885
Queue1   Class2    0.23849  0.17311  1.5086  0.90516  0.15813  0.15811
Queue2   Class1    0.35219  0.22601  1.4366  0.93376  0.24808  0.24811
Queue2   Class2    0.26912  0.18371  1.4685    1.028  0.18432  0.18432
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.2.7 `fcr_oqnwaitq`

The same region with waiting.
The example is built and solved as follows.

```
Finite Capacity Region with Blocking (Multiclass Open Network)
This example creates a multiclass open network with 2 queues and a
finite capacity region with global and per-class constraints.
```

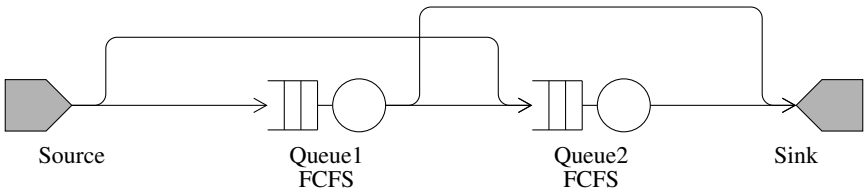

Figure 3.17: Topology of `fcr_oqnwaitq`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.4); Class2: Exp(0.3)
Service	Queue1 – Class1: Exp(1); Class2: Exp(0.9) Queue2 – Class1: Exp(1.1); Class2: Exp(1)
Solvers	JMT

Table 3.17: Features of `fcr_oqnwaitq`.

```
When the region is full, arriving jobs are blocked (wait in queue).
# Create network
model = Network('FCR Blocking Example')
# Add nodes
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Add job classes
class1 = OpenClass(model, 'Class1', 0)
class2 = OpenClass(model, 'Class2', 1)
# Set arrival and service rates
source.set_arrival(class1, Exp.fit_rate(0.4))
source.set_arrival(class2, Exp.fit_rate(0.3))
queue1.set_service(class1, Exp.fit_rate(1.0))
queue1.set_service(class2, Exp.fit_rate(0.9))
queue2.set_service(class1, Exp.fit_rate(1.1))
queue2.set_service(class2, Exp.fit_rate(1.0))
# Create routing matrix
P = model.init_routing_matrix()
# Class1 routing
P.set(class1, class1, source, queue1, 0.5)
P.set(class1, class1, source, queue2, 0.5)

    ... (17 source lines omitted; full implementation: python/examples/advanced/)

fcr.set_drop_rule(class2, False)     # False = block (wait)
# Run JMT (blocking behavior requires longer simulation time in JMT)
solver = JMT(model, seed=23000, samples=10000)
# Get results
avg_table = solver.get_avg_table()
print(avg_table)
```

Running this edition prints

```
Warning [refresh_struct]: Priority classes are specified but no priority-aware scheduling policy (PSPRIO,
     DPSPRIO, GPSPRIO, HOL, FCFSPRIO, FCFSPRPRIO, FCFSPIPRIO, LCFSPRIO, LCFSPRPRIO, LCFSPIPRIO, SRPTPRIO)
     is used in the model. Priorities will be ignored.
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT    ResidT     ArvR     Tput
Source   Class1          0        0        0        0        0  0.39141
Source   Class2          0        0        0        0        0  0.30305
Queue1   Class1    0.31806  0.19559   1.7585  0.87926  0.20234  0.20234
Queue1   Class2    0.31161  0.19379    1.826   1.0956  0.18319  0.18228
Queue2   Class1    0.40681  0.23346   1.6859   1.0959  0.26291   0.2629
Queue2   Class2    0.37782  0.20623   1.7343    1.214  0.21366  0.21036
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by JMT.

3.3 State probabilities

Beyond the averages, the solvers that build a state space can return probabilities. getProb gives the probability of a detailed station state, getProbAggr of an aggregate one (the queue length only, with the phases and the in-service order summed out), and the Sys variants give the joint state of the whole network.

The detailed state is discipline-dependent: under FCFS it records the order of the jobs in service, under PS it does not, and with non-exponential service it carries the phase of each job. That is why the same script exists in an FCFS and a PS version.

3.3.1 statepr_aggr

The marginal probability of an aggregate station state, through getProbAggr.

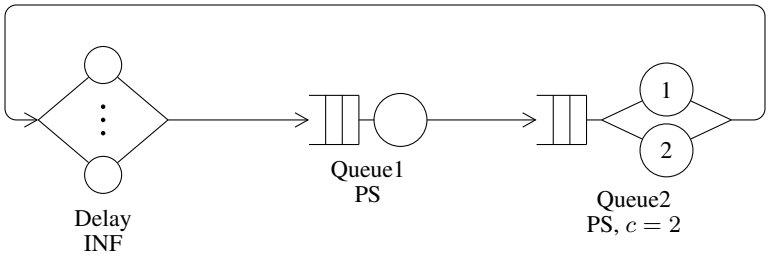

Figure 3.18: Topology of statepr_aggr: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 0$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(3); Class2: Exp(4) Queue2 – Class1: Exp(1); Class2: Exp(3)
Solvers	CTMC, NC

Table 3.18: Features of statepr_aggr.

The example is built and solved as follows.

```
State Probabilities - Aggregated State Space
This example demonstrates how to compute marginal state probabilities for
the aggregated state space using normalizing constants (NC) and CTMC solvers.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network model
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue2.setNumberOfServers(2)
N = [2, 0]
job_class1 = ClosedClass(model, 'Class1', N[0], delay, 0)
job_class2 = ClosedClass(model, 'Class2', N[1], delay, 0)
delay.setService(job_class1, Exp(1))
delay.setService(job_class2, Exp(1))
queue1.setService(job_class1, Exp(3))
queue1.setService(job_class2, Exp(4))
```

```
queue2.setService(job_class1, Exp(1))
queue2.setService(job_class2, Exp(3))
K = model.getNumberOfClasses()
P = [[None, None], [None, None]]
P[0][0] = [[0, 1, 0], [0, 0, 1], [1, 0, 0]]
P[1][1] = [[0, 1, 0], [0, 0, 1], [1, 0, 0]]
model.link(P)
M = model.getNumberOfStations()

    ... (30 source lines omitted; full implementation: python/examples/advanced/)

pr_nc = solver_nc.getProbAggr(target_station)
print(f'NC:   Station {i+1} is in state {target_state} with probability {pr_nc}')
print('\nNote: getProbAggr() computes marginal probabilities for the aggregated')
print('      state space, where each station is specified by a tuple (n_i1, ..., n_iR),')
print('      with n_ir being the number of class r jobs at station i.')
print('      Rows set to [-1, -1] are ignored in the calculation.')
```

Running this edition prints

```
=== State Probabilities - Aggregated State Space ===

This example illustrates the calculation of probabilities via normalizing constants.

Querying probability for station 3 (Queue2) in state [0, 0]

  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
CTMC: Station 3 is in state [0, 0] with probability 0.34
Pmarg_ctmc =
0.34
NC analysis [method: default/cub; type: approximate, deterministic; lang: python; env: python] completed
    in <t>s.
NC:   Station 3 is in state [0, 0] with probability 0.3399999999999999

Note: getProbAggr() computes marginal probabilities for the aggregated
      state space, where each station is specified by a tuple (n_i1, ..., n_iR),
      with n_ir being the number of class r jobs at station i.
      Rows set to [-1, -1] are ignored in the calculation.
```

The rows repeat for each of the 2 solvers the script runs (CTMC, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.3.2 statepr_aggr_large

The same on a model whose state space is large enough to need the aggregate form.

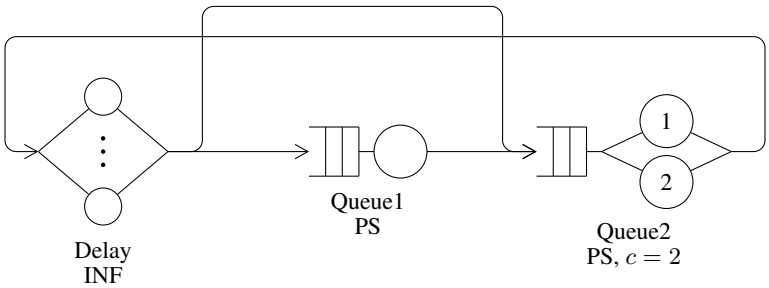

Figure 3.19: Topology of `statepr_aggr_large`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
# Create network
model = Network('model')
```

Nodes	3: Queue × 2, Delay
Classes	4: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 4$), Class4 (closed, $N = 0$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(2); Class3: Exp(1); Class4: Exp(1) Queue1 – Class1: Exp(3); Class2: Exp(4); Class3: Exp(5); Class4: Exp(1) Queue2 – Class1: Exp(1); Class2: Exp(3); Class3: Exp(5); Class4: Exp(2)
Features	class switching on 1 link
Solvers	CTMC, NC

Table 3.19: Features of `statepr_aggr_large`.

```
# Block 1: nodes
node1 = Delay(model, 'Delay')
node2 = Queue(model, 'Queue1', SchedStrategy.PS)
node3 = Queue(model, 'Queue2', SchedStrategy.PS)
node3.setNumberOfServers(2)  # Queue2 has 2 servers
# %%
# Block 2: classes
N = [1, 0, 4, 0]  # Population for each class
jobclass1 = ClosedClass(model, 'Class1', N[0], node1, 0)
jobclass2 = ClosedClass(model, 'Class2', N[1], node1, 0)
jobclass3 = ClosedClass(model, 'Class3', N[2], node1, 0)
jobclass4 = ClosedClass(model, 'Class4', N[3], node1, 0)
# %%
# Set service times for Delay
node1.setService(jobclass1, Exp.fit_mean(1.0))
node1.setService(jobclass2, Exp.fit_mean(1.0/2.0))  # Mean = 0.5
node1.setService(jobclass3, Exp.fit_mean(1.0))
node1.setService(jobclass4, Exp.fit_mean(1.0))
# Set service times for Queue1
node2.setService(jobclass1, Exp.fit_mean(1.0/3.0))  # Mean = 0.333
node2.setService(jobclass2, Exp.fit_mean(1.0/4.0))  # Mean = 0.25

    ... (59 source lines omitted; full implementation: python/examples/advanced/)

# Solve with NC (Normalizing Constant) method
print("\n=== NC Solution (Normalizing Constants) ===")
solver_nc = NC(model, options)
Pr_nc = solver_nc.getProbAggr(node3)
print(f'Station 3 is in state {n[2].tolist()} with probability {Pr_nc}')
print(f'Pr_nc = {Pr_nc}')
```

Running this edition prints

```
  CTMC has 1 SCCs; absorption seeded by the source SCCs
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station 3 is in state [1, 0, 2, 1] with probability 0.0055111404450993925
Pr_ctmc =
0.0055111404450993925

=== NC Solution (Normalizing Constants) ===
NC analysis [method: default/cub; type: approximate, deterministic; lang: python; env: python] completed
    in <t>s.
Station 3 is in state [1, 0, 2, 1] with probability 0.005511140444896116
Pr_nc = 0.005511140444896116
```

The rows repeat for each of the 2 solvers the script runs (CTMC, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.3.3 `statepr_allprobs_fcfs`

The FCFS version, where the detailed state carries the order of the jobs in service.
The example is built and solved as follows.

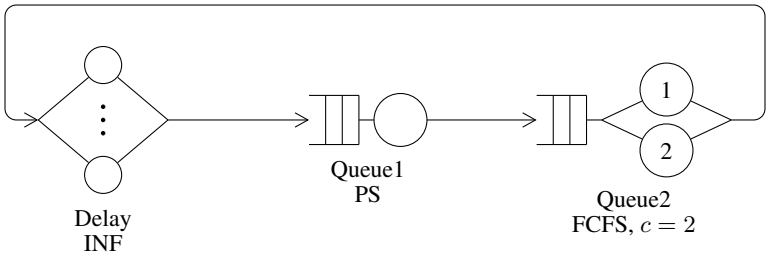

Figure 3.20: Topology of `statepr_allprobs_fcfs`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 0$)
Scheduling	Delay: INF; Queue1: PS; Queue2: FCFS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(3); Class2: Exp(4) Queue2 – Class1: Exp(3); Class2: Exp(3)
Solvers	CTMC, NC, SSA, JMT

Table 3.20: Features of `statepr_allprobs_fcfs`.

```
State Probabilities - All Probability Types (FCFS)
This example demonstrates computation of various types of state probabilities:
- Marginal aggregated probabilities (get_prob_aggr)
- Marginal detailed probabilities (get_prob)
- Joint aggregated probabilities (get_prob_sys_aggr)
- Joint detailed probabilities (get_prob_sys)
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network model with class switching
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
queue2.setNumberOfServers(2)
N = [2, 0]
job_class1 = ClosedClass(model, 'Class1', N[0], delay, 0)
job_class2 = ClosedClass(model, 'Class2', N[1], delay, 0)
delay.setService(job_class1, Exp(1))
delay.setService(job_class2, Exp(1))
queue1.setService(job_class1, Exp(3))
queue1.setService(job_class2, Exp(4))
queue2.setService(job_class1, Exp(3))
queue2.setService(job_class2, Exp(3))
K = model.getNumberOfClasses()
P = model.initRoutingMatrix()
P[0][0] = [[0, 1, 0], [0, 0, 0], [1, 0, 0]]

    ... (70 source lines omitted; full implementation: python/examples/advanced/)

print(f'SSA:  Joint detailed probability = {pr}')
print('\nNote: Aggregated vs. Detailed state spaces:')
print('  - Aggregated: (n_i1, ..., n_iR) - job counts per class at each station')
print('  - Detailed: Also tracks service phases for non-exponential distributions')
print('  - Marginal: Probability for a single station')
print('  - Joint (Sys): Probability for the entire system state')
```

Running this edition prints

```
=== State Probabilities - All Probability Types (FCFS) ===

This example illustrates the calculation of different types of probabilities.

Query state: [[0, 0], [1, 0], [0, 1]]

=== getProbAggr (Marginal Aggregated Probabilities) ===
Probability that station M is in the specified aggregated state.
```

```
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
CTMC: Station 3 is in state [0, 1] with probability 0.3076923076923077
Pmarga_ctmc =
0.3076923076923077
NC analysis [method: default/cub; type: approximate, deterministic; lang: python; env: python] completed
    in <t>s.
NC:   Station 3 is in state [0, 1] with probability 0.30769230769230765

... (37 captured-output lines omitted; rerun the source example for the full output)

SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
SSA:  Joint detailed probability = 0.02389389517380726

Note: Aggregated vs. Detailed state spaces:
  - Aggregated: (n_i1, ..., n_iR) - job counts per class at each station
  - Detailed: Also tracks service phases for non-exponential distributions
  - Marginal: Probability for a single station
  - Joint (Sys): Probability for the entire system state
```

The rows repeat for each of the 4 solvers the script runs (CTMC, NC, SSA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.3.4 `statepr_allprobs_ps`

Every probability API on one PS model: detailed and aggregate, per station and system-wide.

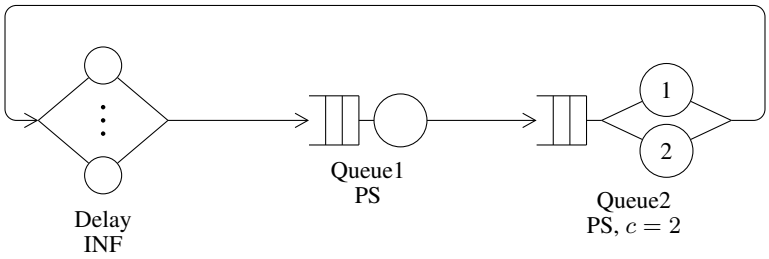

Figure 3.21: Topology of `statepr_allprobs_ps`: a closed network over 3 queueing stations.

Nodes	3: Queue $\times$ 2, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 0$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(3); Class2: Exp(4) Queue2 – Class1: Exp(1); Class2: Exp(3)
Solvers	CTMC, NC, SSA, JMT

Table 3.21: Features of `statepr_allprobs_ps`.

The example is built and solved as follows.

```
State Probabilities - All Probability Types (PS)
This example demonstrates computation of various types of state probabilities
in a processor-sharing queueing network with class switching.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network model with PS queues
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue2.setNumberOfServers(2)
N = [2, 0]
job_class1 = ClosedClass(model, 'Class1', N[0], delay, 0)
```

```
job_class2 = ClosedClass(model, 'Class2', N[1], delay, 0)
delay.setService(job_class1, Exp(1))
delay.setService(job_class2, Exp(1))
queue1.setService(job_class1, Exp(3))
queue1.setService(job_class2, Exp(4))
queue2.setService(job_class1, Exp(1))
queue2.setService(job_class2, Exp(3))
K = model.getNumberOfClasses()
P = model.initRoutingMatrix()
P[0][0] = [[0, 1, 0], [0, 0, 0], [1, 0, 0]]
P[0][1] = [[0, 0, 0], [0, 0, 1], [0, 0, 0]]
P[1][0] = [[0, 1, 0], [0, 0, 0], [1, 0, 0]]
P[1][1] = [[0, 0, 0], [0, 0, 1], [0, 0, 0]]

    ... (66 source lines omitted; full implementation: python/examples/advanced/)

print(f'SSA:  Pjoint = {pr}')
print('\nNote: Probability types:')
print('  - Pmarga:  Marginal probability (aggregated state)')
print('  - Pmarg:   Marginal probability (detailed state with phases)')
print('  - Pjointa: Joint probability (aggregated state)')
print('  - Pjoint:  Joint probability (detailed state with phases)')
```

Running this edition prints

```
=== State Probabilities - All Probability Types (PS) ===

This example illustrates the calculation of different types of probabilities.

Query state: [[0, 0], [1, 0], [0, 1]]

=== getProbAggr (Marginal Aggregated Probabilities) ===
Probability that station M is in the specified aggregated state.

  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
CTMC: Pmarga = 0.3076923076923077
Pmarga_ctmc =
0.3076923076923077
NC analysis [method: default/cub; type: approximate, deterministic; lang: python; env: python] completed
    in <t>s.
NC:   Pmarga = 0.30769230769230765

... (37 captured-output lines omitted; rerun the source example for the full output)

SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
SSA:  Pjoint = 0.02397254890979828

Note: Probability types:
  - Pmarga:  Marginal probability (aggregated state)
  - Pmarg:   Marginal probability (detailed state with phases)
  - Pjointa: Joint probability (aggregated state)
  - Pjoint:  Joint probability (detailed state with phases)
```

The rows repeat for each of the 4 solvers the script runs (CTMC, NC, SSA, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.3.5 statepr_sys_aggr

The joint aggregate state of the whole network, through `getProbSysAggr`.
The example is built and solved as follows.

```
State Probabilities - System-Wide Aggregated Probabilities
This example demonstrates computation of system-wide (joint) state probabilities
for the aggregated state space in a multi-class queueing network with class switching.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network model with 4 classes
model = Network('model')
```

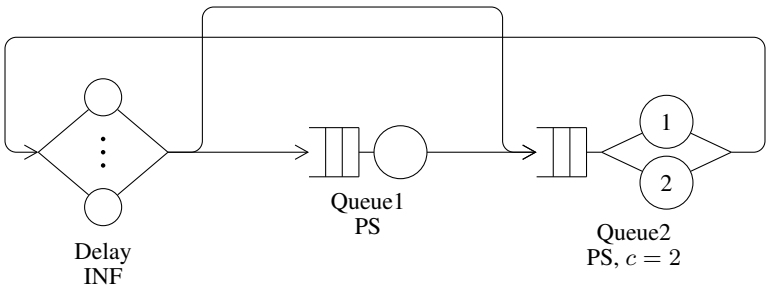

Figure 3.22: Topology of `statepr_sys_aggr`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	4: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 3$), Class4 (closed, $N = 0$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS ($c = 2$)
Service	Delay – Class1: Exp(1); Class2: Exp(2); Class3: Exp(1); Class4: Exp(1) Queue1 – Class1: Exp(3); Class2: Exp(4); Class3: Exp(5); Class4: Exp(1) Queue2 – Class1: Exp(1); Class2: Exp(3); Class3: Exp(5); Class4: Exp(2)
Features	class switching on 1 link
Solvers	CTMC, NC, JMT

Table 3.22: Features of `statepr_sys_aggr`.

```
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue2.setNumberOfServers(2)
N = [1, 0, 3, 0]
job_class1 = ClosedClass(model, 'Class1', N[0], delay, 0)
job_class2 = ClosedClass(model, 'Class2', N[1], delay, 0)
job_class3 = ClosedClass(model, 'Class3', N[2], delay, 0)
job_class4 = ClosedClass(model, 'Class4', N[3], delay, 0)
delay.setService(job_class1, Exp(1))
delay.setService(job_class2, Exp(2))
delay.setService(job_class3, Exp(1))
delay.setService(job_class4, Exp(1))
queue1.setService(job_class1, Exp(3))
queue1.setService(job_class2, Exp(4))
queue1.setService(job_class3, Exp(5))
queue1.setService(job_class4, Exp(1))
queue2.setService(job_class1, Exp(1))
queue2.setService(job_class2, Exp(3))

    ... (50 source lines omitted; full implementation: python/examples/advanced/)

solver_jmt = JMT(model, samples=int(1e4), seed=532733)
pr_jmt = solver_jmt.getProbSysAggr()
print(f'JMT: Pr_jmt = {pr_jmt}')
print('\nNote: getProbSysAggr() returns the joint probability of the entire')
print('      system being in the specified aggregated state.')
print('      This is useful for analyzing the likelihood of specific system configurations.')
```

Running this edition prints

```
=== State Probabilities - System-Wide Aggregated Probabilities ===

This example illustrates the calculation of probabilities via normalizing constants.

Query state: All 4 jobs at station 3 (Queue2)
State configuration: [[0, 0, 0, 0], [0, 0, 0, 0], [1, 0, 3, 0]]

Computing getProbSysAggr() with multiple solvers:

  CTMC has 1 SCCs; absorption seeded from initial state
```

```
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
CTMC: Pr_ctmc = 0.0002626044433948303
Pr_ctmc =
0.0002626044433948303
NC analysis [method: exact/gld; type: approximate, deterministic; lang: python; env: python] completed in
    <t>s.
NC (exact): Pr_nc = 0.0002626044433138508
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 532733
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
JMT: Pr_jmt = 0.0001810469567473483

Note: getProbSysAggr() returns the joint probability of the entire
      system being in the specified aggregated state.
      This is useful for analyzing the likelihood of specific system configurations.
```

The rows repeat for each of the 3 solvers the script runs (CTMC, NC, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.3.6 `statepr_sys_aggr_large`

The large-model version.

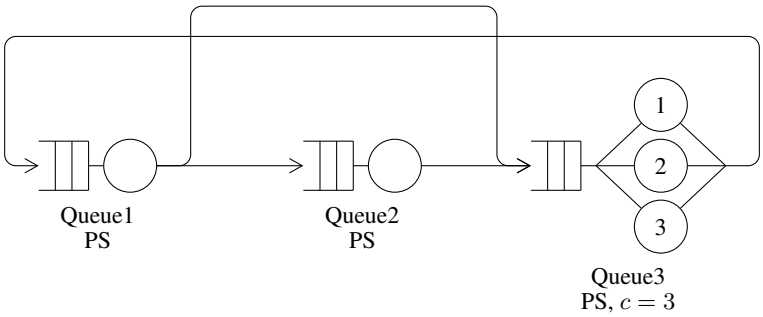

Figure 3.23: Topology of `statepr_sys_aggr_large`: a closed network over 3 queueing stations.

Nodes	3: Queue × 3
Classes	4: Class1 (closed, $N = 1$), Class2 (closed, $N = 1$), Class3 (closed, $N = 1$), Class4 (closed, $N = 1$)
Scheduling	Queue1: PS; Queue2: PS; Queue3: PS ($c = 3$)
Service	Queue1 – Class1: Exp(1); Class2: Exp(2); Class3: Exp(1); Class4: Exp(1) Queue2 – Class1: Exp(3); Class2: Exp(4); Class3: Exp(5); Class4: Exp(1) Queue3 – Class1: Exp(1); Class2: Exp(3); Class3: Exp(5); Class4: Exp(2)
Features	class switching on 1 link
Solvers	CTMC

Table 3.23: Features of `statepr_sys_aggr_large`.

The example is built and solved as follows.

```
State Probabilities - System-Wide Aggregated Probabilities (Large Model)
This example demonstrates computation of system-wide (joint) state probabilities
for a larger multi-class queueing network with class switching.
Expected result from MATLAB: Pr ~= 0.000348
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network model with 4 classes
model = Network('model')
# Block 1: nodes (3 queues, no delay)
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
```

```
queue3 = Queue(model, 'Queue3', SchedStrategy.PS)
queue3.setNumberOfServers(3)
# Block 2: classes
N = [1, 1, 1, 1]
job_class1 = ClosedClass(model, 'Class1', N[0], queue1, 0)
job_class2 = ClosedClass(model, 'Class2', N[1], queue1, 0)
job_class3 = ClosedClass(model, 'Class3', N[2], queue1, 0)
job_class4 = ClosedClass(model, 'Class4', N[3], queue1, 0)
# Set service times for Queue1
queue1.setService(job_class1, Exp(1))
queue1.setService(job_class2, Exp(2))
queue1.setService(job_class3, Exp(1))
queue1.setService(job_class4, Exp(1))
# Set service times for Queue2
queue2.setService(job_class1, Exp(3))

    ... (52 source lines omitted; full implementation: python/examples/advanced/)

print('Pr_ctmc =')
print(pr_ctmc)
print(f'CTMC time: {time.time() - start:.3f}s')
print(f'(Expected MATLAB result: ~0.000348)\n')
print('Note: getProbSysAggr() returns the joint probability of the entire')
print('      system being in the specified aggregated state.')
```

Running this edition prints

```
=== State Probabilities - System-Wide Aggregated (Large Model) ===

This example illustrates the calculation of probabilities via normalizing constants.

Query state: All 4 jobs at station 3 (Queue3)
State configuration: [[0, 0, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1]]

Computing getProbSysAggr() with CTMC solver:

  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
CTMC: Pr_ctmc = 0.0003484356916108198
Pr_ctmc =
0.0003484356916108198
CTMC time: 2.506s
(Expected MATLAB result: ~0.000348)

Note: getProbSysAggr() returns the joint probability of the entire
      system being in the specified aggregated state.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.3.7 `statepr_sys_marg`

The joint law of the per-station TOTAL queue lengths, all classes summed out, through `getProbSysMarg`.

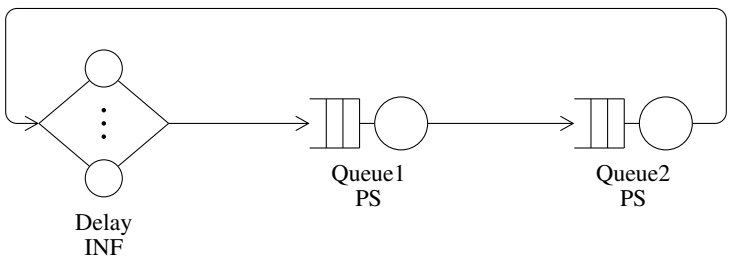

Figure 3.24: Topology of `statepr_sys_marg`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – Class1: Exp(0.6667); Class2: Exp(0.5) Queue1 – Class1: Exp(1.429); Class2: Exp(2.5) Queue2 – Class1: Exp(3.333); Class2: Exp(1.111)
Solvers	CTMC

Table 3.24: Features of `statepr_sys_marg`.

The example is built and solved as follows.

```
State Probabilities - Joint per-station TOTAL queue lengths
Joint probability of the per-station TOTAL queue lengths, all classes summed
out, from SolverNC.getProbSysMarg.
Compare with statepr_sys_aggr.py, which fixes the PER-CLASS population of every
station: each probability here is the sum of that one over every per-class
table with these row sums. The fibre grows combinatorially, so the quantity is
evaluated as a permanent of the demand matrix replicated once per job rather
than by enumerating it.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
N = [2, 1]
job_class1 = ClosedClass(model, 'Class1', N[0], delay, 0)
job_class2 = ClosedClass(model, 'Class2', N[1], delay, 0)
delay.setService(job_class1, Exp(1 / 1.5))
delay.setService(job_class2, Exp(1 / 2.0))
queue1.setService(job_class1, Exp(1 / 0.7))
queue1.setService(job_class2, Exp(1 / 0.4))
queue2.setService(job_class1, Exp(1 / 0.3))
queue2.setService(job_class2, Exp(1 / 0.9))
P = model.initRoutingMatrix()
P.set(job_class1, job_class1, Network.serialRouting(delay, queue1, queue2))
P.set(job_class2, job_class2, Network.serialRouting(delay, queue1, queue2))

    ... (18 source lines omitted; full implementation: python/examples/advanced/)

pb = np.zeros(states.shape[0])
for j in range(states.shape[0]):
    pb[j], _ = solver.getProbSysMarg(states[j, :], 'bethe')
pb = pb / np.sum(pb)
print('  Bethe engine, mean relative error : %.2f%%' % (100 * np.mean(np.abs(pb - p) / p)))
print(solver.citations())
```

Running this edition prints

```
  n(Delay) n(Queue1) n(Queue2)        P(n)
         0         0         3    0.018806
         0         1         2    0.032040
         0         2         1    0.047131
         0         3         0    0.045506
         1         0         2    0.076616
         1         1         1    0.119568
         1         2         0    0.140851
         2         0         1    0.148009
         2         1         0    0.197346
         3         0         0    0.174128
  ----------------------------------------
  sum                              1.000000

  E[n] from the joint law : [1.55012963 0.8614325  0.58843787]
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
  QLen from SolverCTMC    : [1.55012963 0.8614325  0.58843787]
  Bethe engine, mean relative error : 1.58%
[{'key': 'ryse63', 'ref': 'H. J. Ryser, "Combinatorial Mathematics", Carus Mathematical Monographs 14, MAA
```

```
    , 1963', 'covers': 'inclusion-exclusion formula for the exact permanent of the demand matrix'}, {'key
    ': 'vont13', 'ref': 'P. O. Vontobel, "The Bethe Permanent of a Nonnegative Matrix", IEEE Trans.
    Inform. Theory 59(3), 2013', 'covers': 'sum-product approximation of the permanent and its Bethe free
     energy'}]
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.4 Initial state and warm start

Transient analysis needs an initial state. `initFromMarginal` builds one from marginal queue lengths, filling in the phases and the in-service order in a way consistent with the discipline; a state may also be set explicitly. For closed classes the initial state must conserve the population, which is what makes the ordered form necessary rather than convenient.

A simulation can also be started from the steady state computed by another solver, which removes the warm-up transient from the sample path.

3.4.1 `init_state_fcfs_exp`

An initial state set from marginal queue lengths, and the transient it produces on an FCFS network.

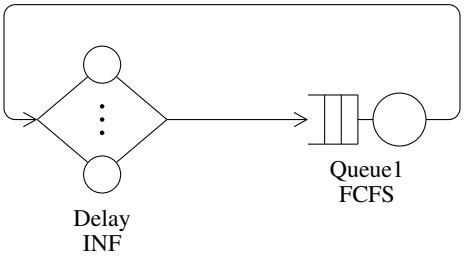

Figure 3.25: Topology of `init_state_fcfs_exp`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: FCFS
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(0.7)

Table 3.25: Features of `init_state_fcfs_exp`.

The example is built and solved as follows.

```
Initial State - FCFS Queue with Exponential Service
This example demonstrates how to specify initial states in a closed queueing
network with FCFS scheduling and exponential service. Different priors on
the initial state distribution affect transient behavior.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with FCFS queue
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
job_class = ClosedClass(model, 'Class1', 5, queue, 0)
delay.setService(job_class, Exp(1))
queue.setService(job_class, Exp(0.7))
M = model.getNumberOfStations()
K = model.getNumberOfClasses()
# Circular routing
P = Network.serial_routing([delay, queue])
```

```
model.link(P)
# Get transient handles
Qt, Ut, Tt = model.getTranHandles()
# Solver options
options = {
    'verbose': 0,
    'samples': int(1e4),
    'stiff': True,
    'timespan': [0, 40]

    ... (31 source lines omitted; full implementation: python/examples/advanced/)

print(f'FLD: {len(QNt_marg_fluid[1][0].t)} time points')
print(f'SteadyStateQLen[CTMC/Prior2]: {QNt_marg_ctmc[1][0].metric[-1]:.6f}')
print(f'SteadyStateQLen[FLD/Prior2]: {QNt_marg_fluid[1][0].metric[-1]:.6f}')
print('\nNote: Different initial state priors lead to different transient behaviors.')
print('      Default initialization starts with all jobs at their reference station.')
print('      Marginal initialization sets a specific number of jobs at each station.')
```

Running this edition prints

```
=== Initial State - FCFS Queue with Exponential Service ===

--- Prior 1: Default initialization ---
Initial state is:
Station 1 (Delay): [0.]
Station 2 (Queue): [1 1 1 1 1]

Transient queue length at station 2, class 1 computed.
CTMC: 401 time points
FLD: 49 time points
SteadyStateQLen[CTMC/Prior1]: 4.300487
SteadyStateQLen[FLD/Prior1]: 4.300000

--- Prior 2: Prior on state with 3 jobs in station 2 ---
Initial state is:
Station 1 (Delay): [2]

... (3 captured-output lines omitted; rerun the source example for the full output)

CTMC: 401 time points
FLD: 49 time points
SteadyStateQLen[CTMC/Prior2]: 4.300487
SteadyStateQLen[FLD/Prior2]: 4.300000

Note: Different initial state priors lead to different transient behaviors.
      Default initialization starts with all jobs at their reference station.
      Marginal initialization sets a specific number of jobs at each station.
```

3.4.2 `init_state_fcfs_nonexp`

The same with non-exponential service, where the phase of the job in service is part of the state.

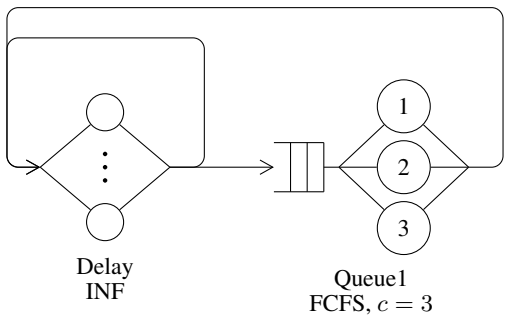

Figure 3.26: Topology of `init_state_fcfs_nonexp`: a closed network over 2 queueing stations.

The example is built and solved as follows.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 3$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: FCFS ($c = 3$)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(1.2); Class2: Erlang(2,2)
Solvers	CTMC, FLD

Table 3.26: Features of `init_state_fcfs_nonexp`.

```
Initial State - FCFS Queue with Non-Exponential Service
This example demonstrates initial state specification with non-exponential
service distributions (Erlang). Shows three different prior distributions
on the initial state.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with FCFS queue and non-exponential service
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue.setNumberOfServers(3)
job_class1 = ClosedClass(model, 'Class1', 3, queue, 0)
job_class2 = ClosedClass(model, 'Class2', 2, queue, 0)
delay.setService(job_class1, Exp(1))
delay.setService(job_class2, Exp(1))
queue.setService(job_class1, Exp(1.2))
queue.setService(job_class2, Erlang.fit_mean_and_scv(1.0, 0.5))
M = model.getNumberOfStations()
K = model.getNumberOfClasses()
# Class-switching routing matrix
P = [[None, None], [None, None]]
P[0][0] = [[0.3, 0.1], [0.2, 0]]
P[0][1] = [[0.6, 0], [0.8, 0]]
P[1][1] = [[0, 1], [0, 0]]
P[1][0] = [[0, 0], [1, 0]]
model.link(P)

    ... (68 source lines omitted; full implementation: python/examples/advanced/)

if len(QNt_fluid_3[1][0].metric) > 0:
    print(f'SteadyStateQLen[FLD/Prior3]: {QNt_fluid_3[1][0].metric[-1]:.6f}')
print('\nNote: This example shows three types of initial state specification:')
print('  1. Default: All jobs at reference stations')
print('  2. Marginal: First state matching the marginal distribution')
print('  3. Uniform: Uniform distribution over all states with given marginal')
```

Running this edition prints

```
=== Initial State - FCFS Queue with Non-Exponential Service ===

--- Prior 1: Default initialization ---
Initial state is:
Station 1 (Delay): [0. 0.]
Station 2 (Queue): [2 2 3 0 0]
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
     <t>s.
CTMC queue length at t=0: 3.0
FLD queue length at t=0: 3.0
SteadyStateQLen[CTMC/Prior1]: 0.179625
SteadyStateQLen[FLD/Prior1]: 0.175029

--- Prior 2: First state with marginal [0,0; 4,1] ---
Initial state is:

... (16 captured-output lines omitted; rerun the source example for the full output)

FLD queue length at t=0: 4.0
SteadyStateQLen[CTMC/Prior3]: 0.181806
SteadyStateQLen[FLD/Prior3]: 0.175861
```

```
Note: This example shows three types of initial state specification:
  1. Default: All jobs at reference stations
  2. Marginal: First state matching the marginal distribution
  3. Uniform: Uniform distribution over all states with given marginal
```

The rows repeat for each of the 2 solvers the script runs (CTMC, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

3.4.3 `init_state_ps`

The processor sharing version.

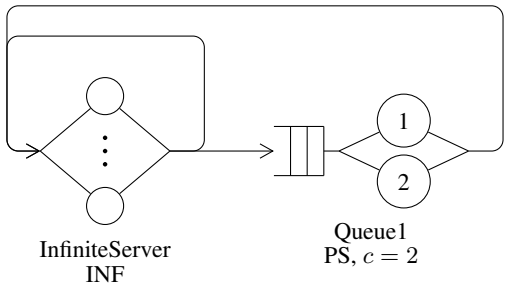

Figure 3.27: Topology of `init_state_ps`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 3$), Class2 (closed, $N = 1$)
Scheduling	InfiniteServer: INF; Queue1: PS ($c = 2$)
Service	InfiniteServer – Class1: Exp(3); Class2: Exp(0.5) Queue1 – Class1: Exp(0.1); Class2: Exp(1)
Solvers	CTMC, JMT, SSA, FLD, MVA, NC

Table 3.27: Features of `init_state_ps`.

The example is built and solved as follows.

```
Initial State - PS Queue with Class Switching
This example demonstrates initial state specification for a processor-sharing
queue with two classes and class-switching behavior. Uses marginal and started
job specification.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with PS queue and class switching
model = Network('model')
delay = Delay(model, 'InfiniteServer')
queue = Queue(model, 'Queue1', SchedStrategy.PS)
queue.setNumberOfServers(2)
job_class1 = ClosedClass(model, 'Class1', 3, delay, 0)
job_class2 = ClosedClass(model, 'Class2', 1, delay, 0)
delay.setService(job_class1, Exp(3))
delay.setService(job_class2, Exp(0.5))
queue.setService(job_class1, Exp(0.1))
queue.setService(job_class2, Exp(1))
M = model.getNumberOfStations()
K = model.getNumberOfClasses()
# Class-switching routing matrix
P = [[None, None], [None, None]]
P[0][0] = [[0.3, 0.1], [0.2, 0]]
P[0][1] = [[0.6, 0], [0.8, 0]]
P[1][1] = [[0, 1], [0, 0]]
P[1][0] = [[0, 0], [1, 0]]
model.link(P)

    ... (39 source lines omitted; full implementation: python/examples/advanced/)
```

```
solver_nc = NC(model, options)
avg_table_nc = solver_nc.getAvgTable()
print(avg_table_nc)
print('\nNote: Initial state affects transient behavior but not steady-state metrics.')
print('      MVA and NC compute steady-state only, so initial state has no effect.')
print('      CTMC, JMT, SSA, and FLD can compute transient behavior from the initial state.')
```

Running this edition prints

```
=== Initial State - PS Queue with Class Switching ===

This example shows solver execution on a 2-class 2-node class-switching model
with specified initial state.

Initial state specification:
  Marginal: [[2,1], [1,0]] - Job distribution across stations and classes
  Started:  [[0,0], [1,0]] - Number of jobs in service at each station

CTMC Solver:
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station         JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
InfiniteServer  Class1    0.33653  0.33653  0.33333  0.19841   1.0096   1.0096
InfiniteServer  Class2      1.373    1.373        2  0.80952  0.68652  0.68652
Queue1          Class1     1.3634  0.50479   13.504  0.80381  0.10096  0.10096

... (45 captured-output lines omitted; rerun the source example for the full output)

InfiniteServer  Class1    0.33653  0.33653  0.33333  0.19841   1.0096   1.0096
InfiniteServer  Class2      1.373    1.373        2  0.80952  0.68652  0.68652
Queue1          Class1     1.3634  0.50479   13.504  0.80381  0.10096  0.10096
Queue1          Class2    0.92708  0.34326   1.3504  0.54659  0.68652  0.68652

Note: Initial state affects transient behavior but not steady-state metrics.
      MVA and NC compute steady-state only, so initial state has no effect.
      CTMC, JMT, SSA, and FLD can compute transient behavior from the initial state.
```

The rows repeat for each of the 6 solvers the script runs (CTMC, JMT, SSA, FLD, MVA, NC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.4.4 `ldes_warmstart`

Seeding a simulation from the steady state of another solver, so the warm-up phase can be skipped.

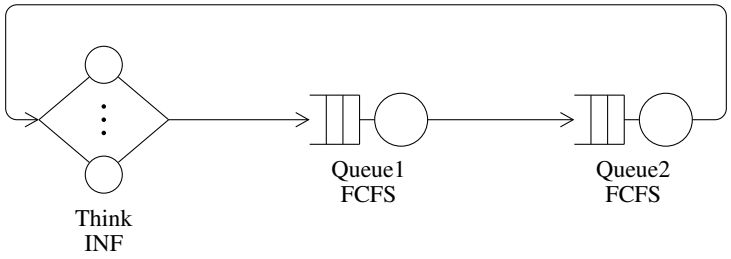

Figure 3.28: Topology of `ldes_warmstart`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	1: Jobs (closed, $N = 100$)
Scheduling	Think: INF; Queue1: FCFS; Queue2: FCFS
Service	Think – Jobs: Exp(1) Queue1 – Jobs: Exp(1) Queue2 – Jobs: Exp(0.98)
Solvers	MVA, CTMC, LDES

Table 3.28: Features of `ldes_warmstart`.

The example is built and solved as follows.

```
                    SchedStrategy, SolverLDES, SolverMVA, SolverCTMC)
"""Closed near-balanced tandem: Think(Exp,1) -> Queue1(Exp,1.0) -> Queue2(Exp,0.98)."""
model = Network('ldesWarmStart')
think = Delay(model, 'Think')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
jobs = ClosedClass(model, 'Jobs', n, think)
think.setService(jobs, Exp(1.0))
queue1.setService(jobs, Exp(1.0))
queue2.setService(jobs, Exp(0.98))  # near-balanced bottleneck
P = model.initRoutingMatrix()
P.set(jobs, jobs, think, queue1, 1.0)
P.set(jobs, jobs, queue1, queue2, 1.0)
P.set(jobs, jobs, queue2, think, 1.0)
model.link(P)
return model
"""Average L1 relative error and total wall-clock over the seed set."""
err = 0.0
rtime = 0.0
for s in seeds:
    model = build_model(n)
    t0 = time.time()
    solver = SolverLDES(model, samples=samples, seed=s)
    if init_sol is not None:
        solver.options.init_sol = init_sol
        solver.options.tranfilter = 'fixed'

    ... (40 source lines omitted; full implementation: python/examples/advanced/)

        warm_at = b
    print('%10d  %7.2f%% %6.2fs  %7.2f%% %6.2fs' % (b, 100 * ce, ct, 100 * we, wt))
    if cold_at is not None and warm_at is not None:
        break
print()
print('Samples to reach 10%% error: COLD=%s, WARM=%s' % (cold_at, warm_at))
```

Running this edition prints

```
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Exact mean queue lengths: [ 0.98 33.71 65.31]
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Warm placement from SolverMVA (0.040s): [ 1. 34. 65.]
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: sync; type: exact, deterministic; lang: python; env: python] completed in <t>s.
CTMC distribution-mode placement (N=20): [ 0.  0. 20.]

   samples     COLD err/time     WARM err/time
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23000
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23001
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23002
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23000 --tranfilter fixed --warmupfrac
      0.0 --initsol 1.0,34.0,65.0
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23001 --tranfilter fixed --warmupfrac
      0.0 --initsol 1.0,34.0,65.0
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 10000 --seed 23002 --tranfilter fixed --warmupfrac
      0.0 --initsol 1.0,34.0,65.0
     10000    58.45%   0.59s    15.51%   0.53s

... (8 captured-output lines omitted; rerun the source example for the full output)

SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23001
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23002
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000 --tranfilter fixed --warmupfrac 0.0 --
     initsol 1.0,34.0,65.0
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23001 --tranfilter fixed --warmupfrac 0.0 --
     initsol 1.0,34.0,65.0
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23002 --tranfilter fixed --warmupfrac 0.0 --
     initsol 1.0,34.0,65.0
    200000     6.17%   3.62s     5.28%   4.12s

Samples to reach 10% error: COLD=200000, WARM=50000
```

The rows repeat for each of the 3 solvers the script runs (MVA, CTMC, LDES), which is the point of

the example: the same model read by methods of different cost and different exactness.

3.5 State-dependent routing

Under RAND or probabilistic routing the destination is drawn from a fixed distribution. State-dependent policies read the network state instead: JSQ sends the job to the shortest queue, k-choices samples k queues and picks the shortest of those, and round-robin and its weighted form cycle deterministically. Because the routing depends on the state, the network is no longer product-form, and JSQ in particular is only available to the simulators.

3.5.1 `sdroute_closed`

The closed counterpart.

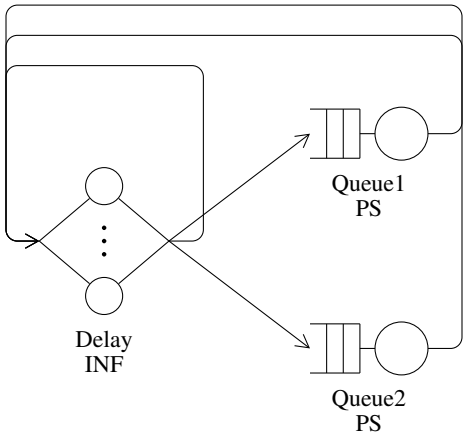

Figure 3.29: Topology of `sdroute_closed`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	1: Class1 (closed, $N = 1$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – Class1: HyperExp(0.99,0.01;1.534,0.02819) Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(2)
Features	class switching on 1 link
Solvers	CTMC, JMT, SSA

Table 3.29: Features of `sdroute_closed`.

The example is built and solved as follows.

```
State-Dependent Routing - Closed Network
This example demonstrates state-dependent routing in a closed queueing network.
A delay node uses round-robin routing to distribute jobs between two queues.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network with state-dependent routing
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
job_class = ClosedClass(model, 'Class1', 1, delay, 0)
delay.setService(job_class, HyperExp.fit_mean_and_scv(1, 25))
queue1.setService(job_class, Exp(1))
queue2.setService(job_class, Exp(2))
# Create links
```

```
model.addLink(delay, delay)
model.addLink(delay, queue1)
model.addLink(delay, queue2)
model.addLink(queue1, delay)
model.addLink(queue2, delay)
# Set state-dependent routing at delay node
delay.setRouting(job_class, RoutingStrategy.RROBIN)
queue1.setProbRouting(job_class, delay, 1.0)
queue2.setProbRouting(job_class, delay, 1.0)
print('=== State-Dependent Routing - Closed Network ===\n')
# Solve with CTMC

    ... (10 source lines omitted; full implementation: python/examples/advanced/)

print('\nSSA (Stochastic State-space Analysis):')
solver_ssa = SSA(model, verbose=True, samples=int(1e4), seed=23000)
avg_table_ssa = solver_ssa.getAvgTable()
print(avg_table_ssa)
print('\nNote: The delay node uses round-robin routing to alternate between Queue1 and Queue2
    .')
print('      Jobs return from queues back to the delay node with probability 1.0.')
```

Running this edition prints

```
=== State-Dependent Routing - Closed Network ===

CTMC (exact):
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util  RespT   ResidT     ArvR     Tput
Delay    Class1    0.66667  0.66667      1        1  0.66667  0.66667
Queue1   Class1    0.22222  0.22222      1  0.33333  0.22222  0.22222
Queue2   Class1    0.11111  0.11111    0.5  0.16667  0.22222  0.22222

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
  Station JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0   Delay   Class1  0.66340  0.66340  0.99642  0.99642  0.67275  0.66595

... (4 captured-output lines omitted; rerun the source example for the full output)

SSA analysis [method: default/nrm; type: approximate, randomized; lang: python; env: python] completed in
    <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.61875  0.61875  0.83313  0.83313   0.7518  0.74269
Queue1   Class1    0.25825  0.25825        1  0.33333  0.24756  0.25825
Queue2   Class1    0.12299  0.12299      0.5  0.16667  0.24756  0.24599

Note: The delay node uses round-robin routing to alternate between Queue1 and Queue2.
      Jobs return from queues back to the delay node with probability 1.0.
```

The rows repeat for each of the 3 solvers the script runs (CTMC, JMT, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

3.5.2 `sdroute_jsq`

Join-the-shortest-queue dispatching over a pool of FCFS queues.
The example is built and solved as follows.

```
State-Dependent Routing - Join the Shortest Queue
This example demonstrates JSQ routing in an open queueing network.
The router sends each arriving job to the queue with the fewest jobs.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
model = Network('myModel')
N = 3
rho = 0.7
```

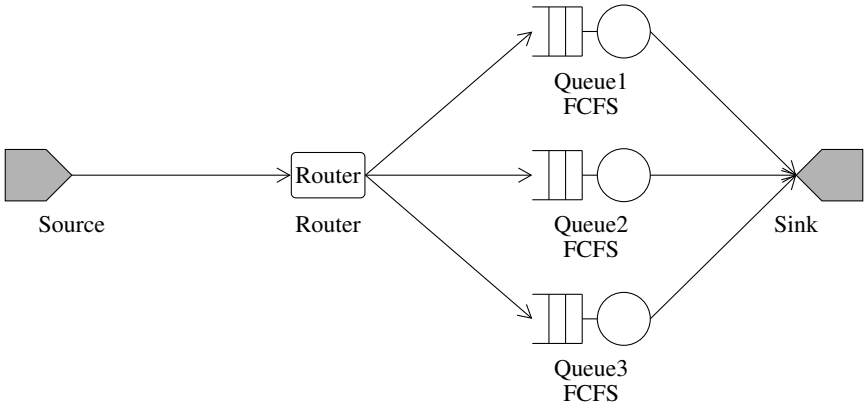

Figure 3.30: Topology of `sdroute_jsq`: an open network over 3 queueing stations with a router.

Nodes	6: Source, Queue × 3, Router, Sink
Classes	1: Class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Arrivals	Source – Class1: Exp(2.1)
Service	Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(1) Queue3 – Class1: Exp(1)
Features	class switching on 6 links
Solvers	JMT, LDES

Table 3.30: Features of `sdroute_jsq`.

```
mu = 1
lam = N * rho * mu
# Block 1: nodes
source = Source(model, 'Source')
router = Router(model, 'Router')
queue = []
for i in range(N):
    queue.append(Queue(model, f'Queue{i+1}', SchedStrategy.FCFS))
sink = Sink(model, 'Sink')
# Block 2: classes
oclass = OpenClass(model, 'Class1')
source.setArrival(oclass, Exp(lam))
for i in range(N):
    queue[i].setService(oclass, Exp(mu))
# Block 3: topology
model.addLink(source, router)
for i in range(N):
    model.addLink(router, queue[i])

    ... (7 source lines omitted; full implementation: python/examples/advanced/)

for solver in solvers:
    print(f'SOLVER: {solver.getName()}')
    avg_table = solver.getAvgNodeTable()
    print(avg_table)
    AvgTable.append(avg_table)
    print()
```

Running this edition prints

```
SOLVER: JMT
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
     Node JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  Source   Class1  0.00000  0.00000  0.00000  0.00000  0.00000  2.14241
1  Queue1   Class1  1.15460  0.69904  1.67728  0.55909  0.70971  0.70330
2  Queue2   Class1  1.19603  0.70625  1.66381  0.55460  0.70313  0.69734
```

```
3  Queue3   Class1  1.14024  0.70621  1.67411  0.55804  0.70054  0.70270
4  Router   Class1  0.00000  0.00000  0.00000  0.00000  2.14241  2.14241
5    Sink   Class1  0.00000  0.00000  0.00000  0.00000  2.14241  0.00000

SOLVER: LDES
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
     Node JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  Source   Class1  0.00000  0.00000  0.00000  0.00000  0.00000  2.10000
1  Router   Class1  0.00000  0.00000  0.00000  0.00000  2.10000  2.10000
2  Queue1   Class1  1.20338  0.70044  1.72414  0.57471  0.69857  0.69785
3  Queue2   Class1  1.20348  0.70088  1.71324  0.57108  0.70306  0.70240
4  Queue3   Class1  1.20648  0.70281  1.72187  0.57396  0.70137  0.70061
5    Sink   Class1  0.00000  0.00000  0.00000  0.00000  2.10000  0.00000
```

The rows repeat for each of the 2 solvers the script runs (JMT, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

3.5.3 `sdroute_krzesinski`

Adaptive routing with balanced loading (Krzesinski 1987): a central server and four single-center branches whose routing probabilities depend on the branch populations, and which keep a product form.

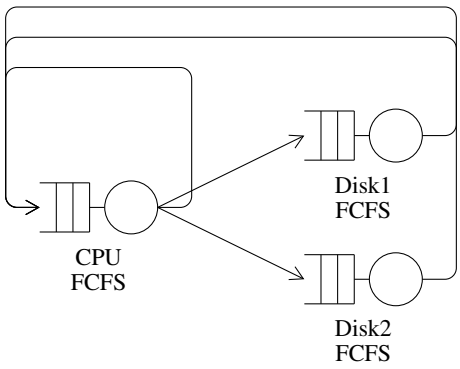

Figure 3.31: Topology of `sdroute_krzesinski`: a closed network over 3 queueing stations.

Nodes	3: Queue × 3
Classes	1: Class1 (closed, $N = 3$)
Scheduling	CPU: FCFS; Disk1: FCFS; Disk2: FCFS
Service	CPU – Class1: Exp(1) Disk1 – Class1: Exp(0.8) Disk2 – Class1: Exp(0.5)
Solvers	NC, CTMC

Table 3.31: Features of `sdroute_krzesinski`.

The example is built and solved as follows.

```
                    SchedStrategy, SolverCTMC, SolverNC)
"""Central server, branches {2} and {3}, C = (-1,-1), d_12=1, d_13=2, d_23=3."""
mu = [1.0, 0.8, 0.5]
N = 3
model = Network('sdr_central_server')
node = [Queue(model, 'CPU', SchedStrategy.FCFS),
        Queue(model, 'Disk1', SchedStrategy.FCFS),
        Queue(model, 'Disk2', SchedStrategy.FCFS)]
jobclass = ClosedClass(model, 'Class1', N, node[0], 0)
for i in range(3):
    node[i].set_service(jobclass, Exp(mu[i]))
model.add_link(node[0], node[0])   # denied entry: the busy form of waiting
model.add_link(node[0], node[1])
model.add_link(node[0], node[2])
model.add_link(node[1], node[0])
model.add_link(node[2], node[0])
```

```
node[1].set_prob_routing(jobclass, node[0], 1.0)
node[2].set_prob_routing(jobclass, node[0], 1.0)
d = np.zeros((2, 3))
d[0, 1] = 1.0
d[0, 2] = 2.0
d[1, 2] = 3.0
node[0].set_state_dep_routing(jobclass, node[0],
                              [[], [node[1]], [node[2]]],
                              [0, 1, 2], [-1, -1], d)
return model
model = krzesinski_sec25()
print(SolverNC(model).get_avg_table())
print(SolverCTMC(model).get_avg_table())
```

Running this edition prints

```
NC analysis [method: sdr; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
CPU      Class1     1.3165  0.74684  1.7627   1.7627  0.99578  0.74684
Disk1    Class1    0.44304  0.44304    1.25  0.41667  0.24895  0.35443
Disk2    Class1     1.2405  0.78481  3.1613   1.0538  0.24895  0.39241
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
CPU      Class1     1.3165  0.74684  1.7627   1.7627  0.99578  0.74684
Disk1    Class1    0.44304  0.44304    1.25  0.41667  0.24895  0.35443
Disk2    Class1     1.2405  0.78481  3.1613   1.0538  0.24895  0.39241
```

The rows repeat for each of the 2 solvers the script runs (NC, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.5.4 `sdroute_multibranch`

The same product form on branches holding several centers, where the coefficients are the branch traffic equations.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	CTMC

Table 3.32: Features of `sdroute_multibranch`.

The example is built and solved as follows.

```
                     SolverCTMC)
"""Branch 2 = B2a -> B2b, with B2b feeding back to B2a w.p. `pback`."""
mu = [1.0, 0.9, 0.7, 0.5]   # centers 1, 2a, 2b, 3
N = 3
model = Network('sdr_multi')
node = [Queue(model, 'CPU', SchedStrategy.FCFS),
        Queue(model, 'B2a', SchedStrategy.FCFS),
        Queue(model, 'B2b', SchedStrategy.FCFS),
        Queue(model, 'B3', SchedStrategy.FCFS)]
jobclass = ClosedClass(model, 'Class1', N, node[0], 0)
for i in range(4):
    node[i].set_service(jobclass, Exp(mu[i]))
model.add_link(node[0], node[0])   # denied entry: the busy form of waiting
model.add_link(node[0], node[1])
model.add_link(node[0], node[3])
model.add_link(node[1], node[2])
model.add_link(node[2], node[0])
model.add_link(node[3], node[0])
node[1].set_prob_routing(jobclass, node[2], 1.0)
if pback > 0:
```

```
    model.add_link(node[2], node[1])
    node[2].set_prob_routing(jobclass, node[1], pback)
    node[2].set_prob_routing(jobclass, node[0], 1.0 - pback)
else:
    node[2].set_prob_routing(jobclass, node[0], 1.0)
node[3].set_prob_routing(jobclass, node[0], 1.0)

    ... (34 source lines omitted; full implementation: python/examples/advanced/)

Q2, X2 = pfqn_sdr(S, xiflat, [N], sn.sdr)[:2]
print('    xi==1 everywhere: max|dQ| = %.3e   max|dX| = %.3e'
      % (np.max(np.abs(Qc - Q2)), np.max(np.abs(Xc - X2))))
print('\n==== SDR with multi-center branches ====')
run_case('A: branch 2 = 2a -> 2b (series)', 0.0)
run_case('B: branch 2 = 2a -> 2b, 2b -> 2a w.p. 0.5 (feedback onto the branch departure)',
    0.5)
```

Running this edition prints

```
==== SDR with multi-center branches ====
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

  A: branch 2 = 2a -> 2b (series)
    xi          = [1. 1. 1. 1.]
    pfqn_sdr Q  = [1.097992 0.416774 0.5591   0.926134]
    CTMC     Q  = [1.097992 0.416774 0.5591   0.926134]
    pfqn_sdr X  = [0.660243 0.318141 0.318141 0.342103]
    CTMC     X  = [0.660243 0.318141 0.318141 0.342103]
    max|dQ| = 6.661e-16   max|dX| = 2.220e-16
    xi==1 everywhere: max|dQ| = 6.661e-16   max|dX| = 2.220e-16
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

  B: branch 2 = 2a -> 2b, 2b -> 2a w.p. 0.5 (feedback onto the branch departure)
    xi          = [1. 2. 2. 1.]
    pfqn_sdr Q  = [0.800493 0.561502 0.763721 0.874284]
    CTMC     Q  = [0.800493 0.561502 0.763721 0.874284]
    pfqn_sdr X  = [0.545258 0.402965 0.402965 0.343775]
    CTMC     X  = [0.545258 0.402965 0.402965 0.343775]
    max|dQ| = 4.441e-16   max|dX| = 1.110e-16
    xi==1 everywhere: max|dQ| = 2.975e-01   max|dX| = 1.150e-01
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.5.5 `sdroute_open`

Round-robin dispatching from a Router node in an open network.

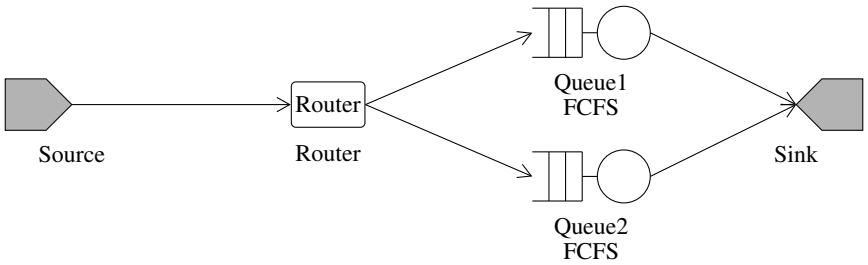

Figure 3.32: Topology of `sdroute_open`: an open network over 2 queueing stations with a router.

The example is built and solved as follows.

```
State-Dependent Routing - Open Network
This example demonstrates state-dependent routing in an open queueing network.
The router uses round-robin routing to alternate between two queues.
```

Nodes	5: Source, Queue × 2, Router, Sink
Classes	1: Class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Queue1 – Class1: Exp(2) Queue2 – Class1: Exp(2)
Features	class switching on 5 links
Solvers	JMT, CTMC

Table 3.33: Features of `sdroute_open`.

```
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create open network with state-dependent routing
model = Network('myModel')
# Block 1: nodes
source = Source(model, 'Source')
router = Router(model, 'Router')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass = OpenClass(model, 'Class1')
source.setArrival(oclass, Exp(1))
queue1.setService(oclass, Exp(2))
queue2.setService(oclass, Exp(2))
# Block 3: topology
model.addLink(source, router)
model.addLink(router, queue1)
model.addLink(router, queue2)
model.addLink(queue1, sink)
model.addLink(queue2, sink)
# Set state-dependent routing: round-robin between Queue1 and Queue2
router.setRouting(oclass, RoutingStrategy.RROBIN)

    ... (8 source lines omitted; full implementation: python/examples/advanced/)

solver_ctmc = CTMC(model, cutoff=5)
avg_table_ctmc = solver_ctmc.getAvgNodeTable()
print(avg_table_ctmc)
print('\nNote: Round-robin routing alternates jobs between Queue1 and Queue2.')
print('      This is state-dependent because routing decisions depend on the')
print('      number of jobs previously routed to each destination.')
```

Running this edition prints

```
=== State-Dependent Routing - Open Network ===

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
     Node JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  Source   Class1  0.00000  0.00000  0.00000  0.00000  0.00000  1.01198
1  Queue1   Class1  0.30240  0.26218  0.57611  0.28806  0.50351  0.50533
2  Queue2   Class1  0.29317  0.25361  0.56951  0.28476  0.50531  0.50719
3  Router   Class1  0.00000  0.00000  0.00000  0.00000  1.01198  1.01198
4    Sink   Class1  0.00000  0.00000  0.00000  0.00000  1.01198  0.00000

CTMC (with cutoff=5):
CTMC solver using state space cutoff = 5 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.

... (6 captured-output lines omitted; rerun the source example for the full output)

1  Router   Class1  0.00000  0.00000  0.00000  0.00000  0.99844  0.99844
2  Queue1   Class1  0.28701  0.24961  0.57492  0.28746  0.49922  0.49922
3  Queue2   Class1  0.28701  0.24961  0.57492  0.28746  0.49922  0.49922
4    Sink   Class1  0.00000  0.00000  0.00000  0.00000  0.99844  0.00000
```

```
Note: Round-robin routing alternates jobs between Queue1 and Queue2.
      This is state-dependent because routing decisions depend on the
      number of jobs previously routed to each destination.
```

The rows repeat for each of the 2 solvers the script runs (JMT, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.5.6 `sdroute_twoclasses_closed`

Two classes with MAP, MMPP(2) and phase-type service under round-robin dispatching.

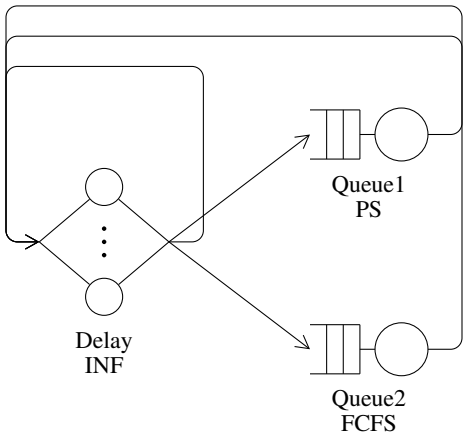

Figure 3.33: Topology of `sdroute_twoclasses_closed`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS; Queue2: FCFS
Service	Delay – Class1: HyperExp(0.99,0.01;1.534,0.02819); Class2: HyperExp(0.99,0.01;1.534,0.02819) Queue1 – Class1: APH; Class2: PH Queue2 – Class1: MAP; Class2: MAP
Features	class switching on 1 link
Solvers	CTMC, JMT, SSA

Table 3.34: Features of `sdroute_twoclasses_closed`.

The example is built and solved as follows.

```
State-Dependent Routing - Closed Network with Two Classes
This example demonstrates state-dependent routing in a closed queueing network
with two job classes. Both classes use round-robin routing at the delay node.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed network with two classes and state-dependent routing
model = Network('model')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
job_class1 = ClosedClass(model, 'Class1', 1, delay, 0)
job_class2 = ClosedClass(model, 'Class2', 2, delay, 0)
# Service distributions for Class 1
# Renewal processes
map21 = APH([1, 0], [[-2, 2], [0, -0.5]])
map22 = PH(1, -1)
# Non-renewal process
map31 = MAP([[-20, 0], [0, -1]], [[0, 20], [0.8, 0.2]])
```

```
map32 = MAP([[-4, 3], [4, -6]], [[1, 0], [0, 2]])
delay.setService(job_class1, HyperExp.fit_mean_and_scv(1, 25))
queue1.setService(job_class1, map21)
queue2.setService(job_class1, map31)
delay.setService(job_class2, HyperExp.fit_mean_and_scv(1, 25))
queue1.setService(job_class2, map22)
queue2.setService(job_class2, map32)
# Create links

    ... (25 source lines omitted; full implementation: python/examples/advanced/)

solver_ssa = SSA(model, verbose=True, samples=int(5e3), seed=23000)
avg_table_ssa = solver_ssa.getAvgTable()
print(avg_table_ssa)
print('\nNote: This example uses various Markovian Arrival Processes (MAP/APH/PH)')
print('      for service distributions, including both renewal and non-renewal processes.')
print('      Both classes use round-robin routing at the delay node.')
```

Running this edition prints

```
=== State-Dependent Routing - Closed Network with Two Classes ===

CTMC (exact):
  CTMC has 439 SCCs; absorption seeded by the source SCCs
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.40007  0.40007        1        1  0.40007  0.40007
Delay    Class2     1.1107   1.1107        1        1   1.1107   1.1107
Queue1   Class1     0.4964  0.33339   3.7223   1.2408  0.13336  0.13336
Queue1   Class2    0.59144  0.37023   1.5975   0.5325  0.37023  0.37023
Queue2   Class1    0.10353  0.07705  0.77637  0.25879  0.13336  0.13336
Queue2   Class2    0.29788  0.25916  0.80459   0.2682  0.37023  0.37023

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000

... (14 captured-output lines omitted; rerun the source example for the full output)

Queue1   Class1    0.51698  0.33684   3.8971    1.299  0.14881  0.13266
Queue1   Class2    0.65499  0.39651   1.6519  0.55063  0.35652  0.39651
Queue2   Class1    0.17572   0.1435   1.2178  0.40595  0.14881  0.14429
Queue2   Class2    0.35583  0.28209  0.88631  0.29544  0.35652  0.40148

Note: This example uses various Markovian Arrival Processes (MAP/APH/PH)
      for service distributions, including both renewal and non-renewal processes.
      Both classes use round-robin routing at the delay node.
```

The rows repeat for each of the 3 solvers the script runs (CTMC, JMT, SSA), which is the point of the example: the same model read by methods of different cost and different exactness.

3.6 Cyclic polling

A polling server visits its queues in a cycle. Under gated service it serves the jobs that were waiting when it arrived, under exhaustive service it stays until the queue empties, and under k-limited service it serves at most k jobs per visit. The discipline is declared as `SchedStrategy.POLLING` and the variant is a parameter of the station.

The polling server parks at a queue rather than roving: the visit order and the ratio between the queue lengths are what a polling model must be validated on, not the aggregate throughput, which is insensitive to the discipline.

3.6.1 `polling_exhaustive_det`

Exhaustive service with deterministic (immediate) switchover.

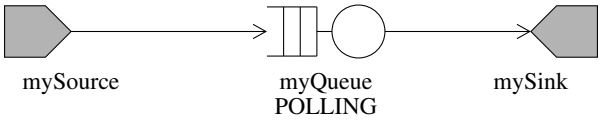

Figure 3.34: Topology of `polling_exhaustive_det`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: POLLING
Arrivals	mySource – myClass1: Det(1); myClass2: Det(1)
Service	myQueue – myClass1: Det(0.001); myClass2: Det(0.001)
Solvers	JMT

Table 3.35: Features of `polling_exhaustive_det`.

The example is built and solved as follows.

```
Cyclic Polling - Exhaustive Service with Deterministic Distributions
This example demonstrates a polling system with exhaustive service policy
using deterministic (constant) arrival and service times.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create M[2]/M[2]/1-Gated polling model
model = Network('M[2]/M[2]/1-Gated')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.POLLING)
sink = Sink(model, 'mySink')
# Block 2: classes with deterministic arrivals and services
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Det.fit_mean(1.0))
queue.setService(oclass1, Det.fit_mean(0.001))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Det.fit_mean(1.0))
queue.setService(oclass2, Det.fit_mean(0.001))
# Set polling policy to exhaustive with immediate switchover
queue.setPollingType(PollingType.EXHAUSTIVE)
queue.setSwitchover(oclass1, Immediate())
queue.setSwitchover(oclass2, Immediate())
# Block 3: topology
P = model.initRoutingMatrix()
P[0] = Network.serial_routing([source, queue, sink])
P[1] = Network.serial_routing([source, queue, sink])

    ... (6 source lines omitted; full implementation: python/examples/advanced/)

# Solve with JMT simulation
print('\nJMT Simulation:')
avg_table_jmt = JMT(model, seed=23000, samples=int(1e5)).getAvgTable()
print(avg_table_jmt)
print('\nNote: This example uses deterministic (constant) arrivals and service times')
print('      with immediate (zero-time) switchovers between queues.')
```

Running this edition prints

```
=== Cyclic Polling - Exhaustive Service (Deterministic) ===

MVA Solution (approximate in general):
Station   JobClass       QLen   Util      RespT     ResidT  ArvR  Tput
mySource  myClass1          0      0          0          0     0     1
mySource  myClass2          0      0          0          0     0     1
myQueue   myClass1  0.0010011  0.001  0.0010011  0.0010011     1     1
myQueue   myClass2  0.0010011  0.001  0.0010011  0.0010011     1     1

JMT Simulation:
JMT Model: <tmp>
```

```
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
    Station  JobClass   QLen   Util  RespT  ResidT  ArvR  Tput
0  mySource  myClass1  0.000  0.000  0.000   0.000   0.0   1.0
1  mySource  myClass2  0.000  0.000  0.000   0.000   0.0   1.0
2   myQueue  myClass1  0.001  0.001  0.001   0.001   1.0   1.0
3   myQueue  myClass2  0.002  0.001  0.002   0.002   1.0   1.0

Note: This example uses deterministic (constant) arrivals and service times
      with immediate (zero-time) switchovers between queues.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by JMT.

3.6.2 polling_exhaustive_exp

Exhaustive service with exponential switchover times.

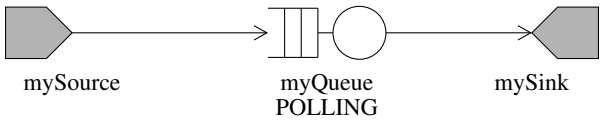

Figure 3.35: Topology of polling_exhaustive_exp: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: POLLING
Arrivals	mySource – myClass1: Exp(0.1); myClass2: Exp(0.1)
Service	myQueue – myClass1: Exp(1); myClass2: Exp(1.5)
Solvers	JMT

Table 3.36: Features of polling_exhaustive_exp.

The example is built and solved as follows.

```
Cyclic Polling - Exhaustive Service with Exponential Distributions
This example demonstrates a polling system with exhaustive service policy,
where the server continues serving a queue until it becomes empty.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create M[2]/M[2]/1-Gated polling model
model = Network('M[2]/M[2]/1-Gated')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.POLLING)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(0.1))
queue.setService(oclass1, Exp(1.0))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.1))
queue.setService(oclass2, Exp(1.5))
# Set polling policy to exhaustive
queue.setPollingType(PollingType.EXHAUSTIVE)
# Block 3: topology
P = model.initRoutingMatrix()
P[0] = Network.serial_routing([source, queue, sink])
P[1] = Network.serial_routing([source, queue, sink])
model.link(P)
print('=== Cyclic Polling - Exhaustive Service (Exponential) ===\n')

    ... (4 source lines omitted; full implementation: python/examples/advanced/)
```

```
# Solve with JMT simulation
print('\nJMT Simulation:')
avg_table_jmt = JMT(model, seed=23000).getAvgTable()
print(avg_table_jmt)
print('\nNote: In exhaustive polling, the server continues serving jobs from a queue')
print('      until it becomes empty before switching to the next queue.')
```

Running this edition prints

```
=== Cyclic Polling - Exhaustive Service (Exponential) ===

MVA Solution (approximate in general):
Station   JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
mySource  myClass1         0         0        0        0     0   0.1
mySource  myClass2         0         0        0        0     0   0.1
myQueue   myClass1   0.11685       0.1   1.1685   1.1685   0.1   0.1
myQueue   myClass2  0.084724  0.066667  0.84724  0.84724   0.1   0.1

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
    Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  mySource  myClass1  0.00000  0.00000  0.00000  0.00000  0.00000  0.09970
1  mySource  myClass2  0.00000  0.00000  0.00000  0.00000  0.00000  0.10064
2   myQueue  myClass1  0.11377  0.09944  1.15023  1.15023  0.09970  0.09923
3   myQueue  myClass2  0.08284  0.06464  0.83470  0.83470  0.10064  0.09965

Note: In exhaustive polling, the server continues serving jobs from a queue
      until it becomes empty before switching to the next queue.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.6.3 `polling_gated`

A gated polling server: at each visit it serves the jobs that were waiting when it arrived.

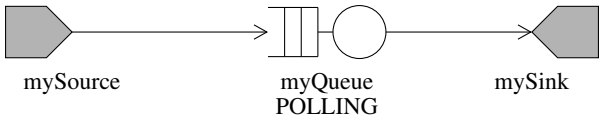

Figure 3.36: Topology of `polling_gated`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: POLLING
Arrivals	mySource – myClass1: Exp(1); myClass2: Exp(0.8)
Service	myQueue – myClass1: Exp(4); myClass2: Exp(1.5)
Solvers	JMT

Table 3.37: Features of `polling_gated`.

The example is built and solved as follows.

```
Cyclic Polling - Gated Service Policy
This example demonstrates a polling system with gated service policy,
where the server serves only those jobs present when polling begins.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create M[2]/M[2]/1-Exhaustive polling model with gated policy
model = Network('M[2]/M[2]/1-Exhaustive')
# Block 1: nodes
```

```
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.POLLING)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(1.0))
queue.setService(oclass1, Exp(4.0))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.8))
queue.setService(oclass2, Exp(1.5))
# Set polling policy to gated with exponential switchover times
queue.setPollingType(PollingType.GATED)
queue.setSwitchover(oclass1, Exp(1.0))
queue.setSwitchover(oclass2, Exp(0.5))
# Block 3: topology
P = model.initRoutingMatrix()
P[0] = Network.serial_routing([source, queue, sink])
P[1] = Network.serial_routing([source, queue, sink])

    ... (7 source lines omitted; full implementation: python/examples/advanced/)

print('\nJMT Simulation:')
avg_table_jmt = JMT(model, seed=23000, samples=int(1e6)).getAvgTable()
print(avg_table_jmt)
print('\nNote: In gated polling, the server serves only jobs present at the queue')
print('      when it begins polling. Jobs arriving during service must wait for')
print('      the next polling cycle.')
```

Running this edition prints

```
=== Cyclic Polling - Gated Service Policy ===

MVA Solution (exact):
Station   JobClass    QLen     Util   RespT  ResidT  ArvR  Tput
mySource  myClass1       0        0       0       0     0     1
mySource  myClass2       0        0       0       0     0   0.8
myQueue   myClass1   11.22     0.25   11.22   11.22     1     1
myQueue   myClass2  11.403  0.53333  14.254  14.254   0.8   0.8

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
    Station  JobClass      QLen     Util     RespT    ResidT     ArvR     Tput
0  mySource  myClass1   0.00000  0.00000   0.00000   0.00000  0.00000  1.00123
1  mySource  myClass2   0.00000  0.00000   0.00000   0.00000  0.00000  0.80026
2   myQueue  myClass1  11.31003  0.24989  11.33990  11.33990  1.00123  1.00080
3   myQueue  myClass2  11.31180  0.53337  14.14869  14.14869  0.80026  0.79995

Note: In gated polling, the server serves only jobs present at the queue
      when it begins polling. Jobs arriving during service must wait for
      the next polling cycle.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.6.4 `polling_klimited`

k-limited service, at most k jobs per visit.

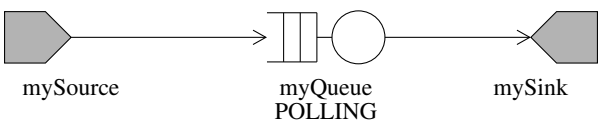

Figure 3.37: Topology of `polling_klimited`: an open network over 1 queueing station.

The example is built and solved as follows.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: POLLING
Arrivals	mySource – myClass1: Exp(0.2); myClass2: Exp(0.3)
Service	myQueue – myClass1: Exp(1); myClass2: Exp(1.5)
Solvers	JMT

Table 3.38: Features of `polling_klimited`.

```
Cyclic Polling - K-Limited Service Policy
This example demonstrates a polling system with k-limited service policy,
where the server serves at most k jobs from a queue before switching.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create M[2]/M[2]/1-Gated polling model with k-limited policy
model = Network('M[2]/M[2]/1-Gated')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.POLLING)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(0.2))
queue.setService(oclass1, Exp(1.0))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.3))
queue.setService(oclass2, Exp(1.5))
# Set polling policy to k-limited with k=1 (serve at most 1 job per visit)
queue.setPollingType(PollingType.KLIMITED, 1)
queue.setSwitchover(oclass1, Exp(1))
queue.setSwitchover(oclass2, Immediate())
# Block 3: topology
P = model.initRoutingMatrix()
P[0] = Network.serial_routing([source, queue, sink])
P[1] = Network.serial_routing([source, queue, sink])

    ... (7 source lines omitted; full implementation: python/examples/advanced/)

print('\nJMT Simulation:')
avg_table_jmt = JMT(model, samples=int(1e5), seed=23000).getAvgTable()
print(avg_table_jmt)
print('\nNote: In k-limited polling with k=1, the server serves at most 1 job')
print('      from each queue before switching to the next queue.')
print('      This prevents a busy queue from monopolizing the server.')
```

Running this edition prints

```
=== Cyclic Polling - K-Limited Service Policy ===

MVA Solution (approximate):
Station   JobClass     QLen  Util   RespT  ResidT  ArvR  Tput
mySource  myClass1        0     0       0       0     0   0.2
mySource  myClass2        0     0       0       0     0   0.3
myQueue   myClass1  0.66667   0.2  3.3333  3.3333   0.2   0.2
myQueue   myClass2   1.1333   0.2  3.7778  3.7778   0.3   0.3

JMT Simulation:
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
    Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0  mySource  myClass1  0.00000  0.00000  0.00000  0.00000  0.00000  0.19924
1  mySource  myClass2  0.00000  0.00000  0.00000  0.00000  0.00000  0.30098
2   myQueue  myClass1  0.78887  0.19856  3.97951  3.97951  0.19924  0.20042
3   myQueue  myClass2  1.39093  0.20010  4.73925  4.73925  0.30098  0.30096

Note: In k-limited polling with k=1, the server serves at most 1 job
      from each queue before switching to the next queue.
```

```
    This prevents a busy queue from monopolizing the server.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.7 Switchover times

Moving the server between queues costs time. A switchover distribution is declared per arc of the polling cycle, and its mean adds directly to the cycle time, so the waiting time grows even though no extra work is done.

3.7.1 `switchover_basic`

Erlang switchover times between the queues of a polling server, and their cost in response time.

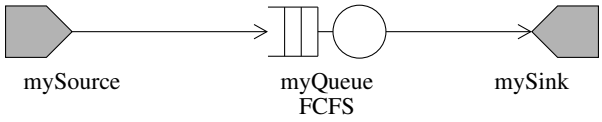

Figure 3.38: Topology of `switchover_basic`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass1: Exp(0.2); myClass2: Exp(0.8)
Service	myQueue – myClass1: Exp(0.1); myClass2: Exp(1.5)
Solvers	JMT

Table 3.39: Features of `switchover_basic`.

The example is built and solved as follows.

```
# Create M[2]/M[2]/1-Gated model with switchover times
model = Network('M[2]/M[2]/1-Gated')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(0.2))
queue.setService(oclass1, Exp(0.1))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.8))
queue.setService(oclass2, Exp(1.5))
# Set switchover times between classes
# Switchover from Class1 to Class2: Exponential with rate 1
queue.setSwitchover(oclass1, oclass2, Exp(1))
# Switchover from Class2 to Class1: Erlang with mean 1 and order 2
queue.setSwitchover(oclass2, oclass1, Erlang(1, 2))
# Block 3: topology
P = model.initRoutingMatrix()
P[0] = Network.serial_routing([source, queue, sink])
P[1] = Network.serial_routing([source, queue, sink])
model.link(P)
return model
model = switchover_basic()

print('=== Switchover Times - Basic Example ===\n')
```

```
# Solve with JMT simulation
print('JMT Simulation:')
avg_table_jmt = JMT(model, seed=23000, keep=True).get_avg_table()
print(avg_table_jmt)
```

Running this edition prints

```
=== Switchover Times - Basic Example ===

JMT Simulation:
python/line_solver/api/solvers/jmt/handler.py:2858: UserWarning: JMT does not support switchover times for
      non-polling queues. Switchover times will be ignored for myQueue.
  warnings.warn(f"JMT does not support switchover times for non-polling queues. "
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Station   JobClass    QLen     Util    RespT  ResidT     ArvR      Tput
mySource  myClass1       0        0        0       0        0   0.19599
mySource  myClass2       0        0        0       0        0   0.81958
myQueue   myClass1  2210.4  0.78376    38679   38679  0.19599  0.078203
myQueue   myClass2  2267.8  0.20632  9851.9  9851.9  0.81958   0.32287
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.8 Response time distribution

`getCdfRespT` returns the cumulative distribution of the response time, per station and per class, rather than its mean. The distribution is what a percentile service level objective is written against, and it is far more sensitive to the service distribution than the mean is: two models with the same mean response time can have very different tails.

3.8.1 `cdf_respt_closed`

The response time CDF at a station of a closed PS network.

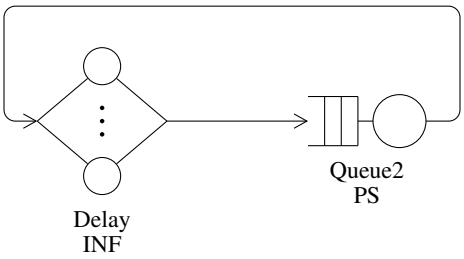

Figure 3.39: Topology of `cdf_respt_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 1$)
Scheduling	Delay: INF; Queue2: PS
Service	Delay – Class1: Exp(10) Queue2 – Class1: Erlang(3,3)
Solvers	JMT, FLD

Table 3.40: Features of `cdf_respt_closed`.

The example is built and solved as follows.

```
CDF of Response Time - Closed Network
```

```
This example demonstrates how to compute the cumulative distribution function (CDF)
of response times in a closed queueing network using both JMT simulation and Fluid solver.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with 2 stations
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue2', SchedStrategy.PS)
job_class = ClosedClass(model, 'Class1', 1, delay, 0)
# Set service distributions
serv_proc1 = Exp(1/0.1)
delay.setService(job_class, serv_proc1)
serv_proc2 = Erlang.fit_mean_and_scv(1, 1/3)
queue.setService(job_class, serv_proc2)
# Serial routing between stations
model.link(Network.serial_routing([delay, queue]))
print('=== CDF Response Time - Closed Network ===\n')
# Solve with JMT simulation
print('Computing CDF with JMT simulation...')
solver_jmt = JMT(model, seed=23000, samples=10000)
rd_sim = solver_jmt.get_cdf_resp_t()
# Compute statistics from CDF (simulation)
avg_respt_from_cdf_sim = []
sq_coeff_of_variation_resp_t_from_cdf_sim = []
for i in range(model.getNumberOfStations()):

    ... (64 source lines omitted; full implementation: python/examples/advanced/)

print('\nSquared Coefficient of Variation from Theory:')
print(sq_coeff_of_variation_resp_t_from_theory)
print('\nSquared Coefficient of Variation from CDF (Simulation):')
print(sq_coeff_of_variation_resp_t_from_cdf_sim)
print('\nSquared Coefficient of Variation from CDF (Fluid):')
print(sq_coeff_of_variation_resp_t_from_cdf_fluid)
```

Running this edition prints

```
=== CDF Response Time - Closed Network ===

Computing CDF with JMT simulation...
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.
Computing CDF with Fluid solver...
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

Since there is a single job, mean and squared coefficient of variation
of response times are close, up to fluid approximation precision, to those
of the service time distribution.


... (9 captured-output lines omitted; rerun the source example for the full output)

Squared Coefficient of Variation from Theory:
[1.0, 0.3333333333333333]

Squared Coefficient of Variation from CDF (Simulation):
[[np.float64(1.0110174517844117)], [np.float64(0.33645388652735425)]]

Squared Coefficient of Variation from CDF (Fluid):
[[np.float64(1.0003874894026954)], [np.float64(0.3349066756161576)]]
```

The rows repeat for each of the 2 solvers the script runs (JMT, FLD), which is the point of the example: the same model read by methods of different cost and different exactness.

3.8.2 cdf_respt_closed_threeclasses

The three-class version, with Erlang service.

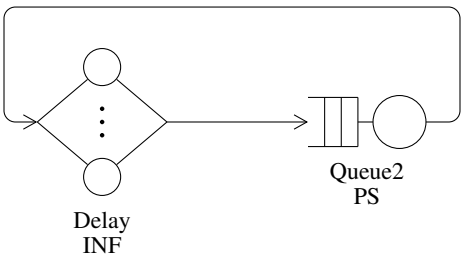

Figure 3.40: Topology of cdf_respt_closed_threeclasses: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	3: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 0$)
Scheduling	Delay: INF; Queue2: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1) Queue2 – Class1: Exp(1); Class2: Erlang(0.5,2); Class3: Exp(100)
Solvers	FLD

Table 3.41: Features of cdf_respt_closed_threeclasses.

The example is built and solved as follows.

```
CDF of Response Time - Closed Network with Three Classes
This example demonstrates response time CDF computation for a closed network
with three job classes that have class-switching behavior.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with 3 classes
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue2', SchedStrategy.PS)
# Class definitions
job_class1 = ClosedClass(model, 'Class1', 1, delay, 0)
job_class2 = ClosedClass(model, 'Class2', 0, delay, 0)
job_class3 = ClosedClass(model, 'Class3', 0, delay, 0)
# Disable completion for class 1 to maintain population
job_class1.completes = False
# Set service times
delay.setService(job_class1, Exp(1/1))
delay.setService(job_class2, Exp(1/1))
delay.setService(job_class3, Exp(1/1))
queue.setService(job_class1, Exp(1/1))
queue.setService(job_class2, Erlang(1/2, 2))
queue.setService(job_class3, Exp(1/0.01))
M = model.getNumberOfStations()
K = model.getNumberOfClasses()
# Routing matrix with class switching
P = [[None for _ in range(K)] for _ in range(K)]

    ... (46 source lines omitted; full implementation: python/examples/advanced/)

print('\nSquared Coefficient of Variation from CDF:')
print(sq_coeff_of_variation_resp_t_from_cdf)
print('\nNote: This example demonstrates class-switching behavior:')
print('  - Class 1 jobs switch to Class 2 at the queue')
print('  - Class 2 jobs switch back to Class 1 at the delay')
print('  - Class 3 jobs follow a simple circular route without switching')
```

Running this edition prints

```
=== CDF Response Time - Closed Network with Three Classes ===

Computing with Fluid solver (state-dependent method)...
Fluid analysis [method: matrix; type: approximate, deterministic; lang: python; env: python] completed in
     <t>s.

Average Response Time:
[[1. 1. 0.]
 [1. 4. 0.]]

Average Response Time from CDF:
[[np.float64(1.0021277378629745), np.float64(1.0021277378629745), 0], [np.float64(1.0021277378629745), np.
     float64(4.000776344330656), 0]]

Squared Coefficient of Variation from CDF:
[[np.float64(1.0022703984784844), np.float64(1.0022703984784844), 0], [np.float64(1.0022703984784844), np.
     float64(0.5012011227458892), 0]]

Note: This example demonstrates class-switching behavior:
  - Class 1 jobs switch to Class 2 at the queue
  - Class 2 jobs switch back to Class 1 at the delay
  - Class 3 jobs follow a simple circular route without switching
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `FLD`.

3.8.3 `cdf_respt_distrib`

The distribution object returned by `getCdfRespT` and the percentiles read from it.

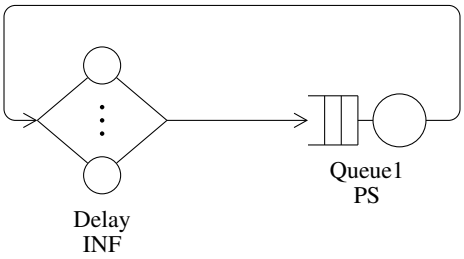

Figure 3.41: Topology of `cdf_respt_distrib`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 3$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Erlang(0.5,2) Queue1 – Class1: Exp(0.5); Class2: HyperExp(0.99,0.01;0.324,0.005143)
Solvers	FLD, JMT

Table 3.42: Features of `cdf_respt_distrib`.

The example is built and solved as follows.

```
CDF of Response Time - Different Service Distributions
This example shows how response time CDFs vary with different service time
distributions (Exp, Erlang, HyperExp) in a closed queueing network.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create closed model with 2 classes
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.PS)
# Class 1: 1 job with exponential service
job_class1 = ClosedClass(model, 'Class1', 1, delay, 0)
delay.setService(job_class1, Exp.fit_mean(1.0))
```

```
queue.setService(job_class1, Exp.fit_mean(2.0))
# Class 2: 3 jobs with non-exponential service distributions
job_class2 = ClosedClass(model, 'Class2', 3, delay, 0)
delay.setService(job_class2, Erlang.fit_mean_and_order(4.0, 2))
queue.setService(job_class2, HyperExp.fit_mean_and_scv(5.0, 30.0))
# Routing matrix
P = [[None, None], [None, None]]
P[0][0] = [[0, 1], [1, 0]]  # circul(2)
P[0][1] = [[0, 0], [0, 0]]
P[1][0] = [[0, 0], [0, 0]]
P[1][1] = [[0, 1], [1, 0]]  # circul(2)
model.link(P)
print('=== CDF Response Time - Different Service Distributions ===\n')
# Solve with Fluid solver

    ... (46 source lines omitted; full implementation: python/examples/advanced/)

print('\nService Processes by Station and Class:')
for i, procs in enumerate(service_procs):
    print(f'Station {i}: {procs}')
print('\nNote: Class 1 uses Exponential service distributions.')
print('      Class 2 uses Erlang (delay) and HyperExponential (queue) distributions.')
print('      This shows how different service distributions affect response time CDFs.')
```

Running this edition prints

```
=== CDF Response Time - Different Service Distributions ===

Computing CDF with Fluid solver (steady-state)...
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
    completed in <t>s.
Computing transient CDF with JMT simulation...
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
    <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.

Average Response Time from CDF (Fluid):
[[np.float64(1.0021083677788074), np.float64(4.001931871344985)], [np.float64(6.574727593224021), np.
    float64(9.358366422838554)]]

Average Response Time from CDF (Simulation):
[[np.float64(1.0004927175088867), np.float64(3.989836128155989)], [np.float64(6.432753573608269), np.
    float64(16.53085952272273)]]

Service Processes by Station and Class:
Station 0: ['N/A', 'N/A']
Station 1: ['N/A', 'N/A']

Note: Class 1 uses Exponential service distributions.
      Class 2 uses Erlang (delay) and HyperExponential (queue) distributions.
      This shows how different service distributions affect response time CDFs.
```

The rows repeat for each of the 2 solvers the script runs (FLD, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.8.4 `cdf_respt_open_twoclasses`

The open two-class version, on FCFS stations.

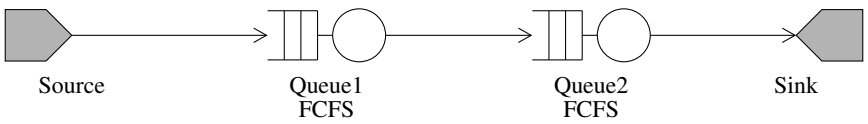

Figure 3.42: Topology of `cdf_respt_open_twoclasses`: an open network over 2 queueing stations.

The example is built and solved as follows.

Nodes	4: Source, Queue × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.25); Class2: Exp(0.25)
Service	Queue1 – Class1: Exp(1); Class2: Exp(1) Queue2 – Class1: Exp(1); Class2: Exp(1)
Solvers	FLD, JMT

Table 3.43: Features of `cdf_respt_open_twoclasses`.

```
CDF of Response Time - Open Network with Two Classes
This example demonstrates response time CDF computation for an open network
with two job classes using both Fluid and JMT solvers.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Create open network
model = Network('myModel')
# Block 1: nodes
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
job_class1 = OpenClass(model, 'Class1', 0)
job_class2 = OpenClass(model, 'Class2', 0)
source.setArrival(job_class1, Exp.fit_mean(4.0))
source.setArrival(job_class2, Exp.fit_mean(4.0))
queue1.setService(job_class1, Exp.fit_mean(1.0))
queue1.setService(job_class2, Exp.fit_mean(1.0))
queue2.setService(job_class1, Exp.fit_mean(1.0))
queue2.setService(job_class2, Exp.fit_mean(1.0))
# Block 3: topology
# MATLAB routing: P{r,s}(i,j) means job in class r at node i goes to class s at node j
P = model.init_routing_matrix()
P.set(job_class1, job_class1, source, queue1, 1)     # P{1,1}(1,2) = 1
P.set(job_class1, job_class2, queue1, queue2, 1)     # P{1,2}(2,3) = 1

    ... (39 source lines omitted; full implementation: python/examples/advanced/)

print('\nAverage Response Time from CDF (Simulation):')
for i, station_avg in enumerate(avg_respt_from_cdf_sim):
    if i > 0:  # Skip source
        print(f'Station {i}: {station_avg}')
print('\nNote: Response time tail distributions are available in rd_fluid and rd_sim')
print('      for detailed analysis and plotting.')
```

Running this edition prints

```
=== CDF Response Time - Open Network with Two Classes ===

Computing CDF with Fluid solver...
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Computing transient CDF with JMT simulation...
JMT Model: <tmp>
SolverJMT command: /usr/lib/jvm/java-17-openjdk-amd64/bin/java -cp common/JMT.jar jmt.commandline.Jmt sim
     <tmp> -seed 23000
JMT analysis [method: jsim; type: approximate, randomized; lang: python; env: python] completed in <t>s.

Average Response Time from CDF (Fluid):
Station 1: [np.float64(1.5026356600914863), np.float64(1.5026356600914843)]
Station 2: [np.float64(1.5026356600914876), np.float64(1.502635660091484)]

Average Response Time from CDF (Simulation):
Station 1: [np.float64(1.9357080518175342), np.float64(1.9501747854016953)]
Station 2: [np.float64(2.0903940502862093), np.float64(2.068766301164727)]

Note: Response time tail distributions are available in rd_fluid and rd_sim
      for detailed analysis and plotting.
```

The rows repeat for each of the 2 solvers the script runs (FLD, JMT), which is the point of the example: the same model read by methods of different cost and different exactness.

3.8.5 cdf_respt_populations

How the response time distribution moves as the closed population grows.

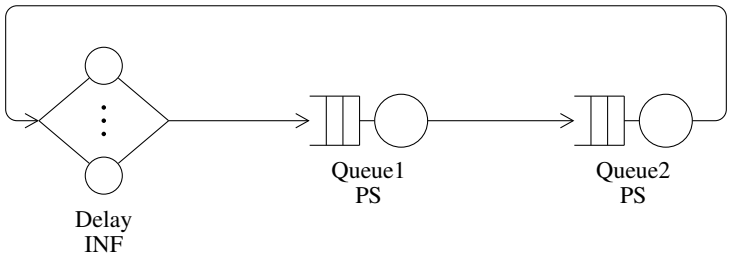

Figure 3.43: Topology of cdf_respt_populations: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	1: Class1 (closed, $N = 8$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(0.5) Queue2 – Class1: Exp(0.5)
Solvers	FLD

Table 3.44: Features of cdf_respt_populations.

The example is built and solved as follows.

```
CDF of Response Time - Varying Population Sizes
This example demonstrates how response time CDFs change as the number
of jobs in a closed network increases.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Test with different population sizes
n_jobs_list = [1, 4, 8]
print('=== CDF Response Time - Varying Population Sizes ===\n')
results = []
for N in n_jobs_list:
    print(f'\n--- Population N = {N} jobs ---')
    model = Network('model')
    delay = Delay(model, 'Delay')
    queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
    queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
    job_class = ClosedClass(model, 'Class1', N, delay, 0)
    job_class.completes = False
    delay.setService(job_class, Exp(1/1))
    queue1.setService(job_class, Exp(1/2))
    queue2.setService(job_class, Exp(1/2))
    # Circular routing through all stations (circul(M))
    # circul(3) = [[0,1,0], [0,0,1], [1,0,0]]
    model.link([[0, 1, 0], [0, 0, 1], [1, 0, 0]])
    # Solve with Fluid solver
    solver = FLD(model, {'iter_max': 100})
    avg_respt = solver.get_avg_respt()

    ... (36 source lines omitted; full implementation: python/examples/advanced/)

print('\n\n=== Summary ===')
print('\nAs population increases, response times increase due to queueing effects.')
print('The CDF shifts to the right, showing higher probability of longer response times.')
print('\nResponse time statistics for each population:')
for result in results:
    print(f"\nN = {result['N']}: Mean Response Times = {result['avg_respt_from_cdf']}")
```

Running this edition prints

```
=== CDF Response Time - Varying Population Sizes ===

--- Population N = 1 jobs ---
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Average Response Time: [[1.]
 [2.]
 [2.]]
FLD getAvgRespT N=1: 1.000000 2.000000 2.000000
Average Response Time from CDF: [[np.float64(1.001858370255721), np.float64(2.002737905498945), np.float64
     (2.002737905498386)]]
Squared Coefficient of Variation from CDF: [[np.float64(1.0046149635541546), np.float64
     (1.0044771463158928), np.float64(1.0044771463227653)]]

--- Population N = 4 jobs ---
Fluid analysis [method: default/dae; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Average Response Time: [[1.        ]
 [4.5175131 ]
 [4.51751292]]

... (17 captured-output lines omitted; rerun the source example for the full output)


Response time statistics for each population:

N = 1: Mean Response Times = [[np.float64(1.001858370255721), np.float64(2.002737905498945), np.float64
     (2.002737905498386)]]

N = 4: Mean Response Times = [[np.float64(1.0013695215372485), np.float64(4.521483996536756), np.float64
     (4.521483975840001)]]

N = 8: Mean Response Times = [[np.float64(1.0013333701362015), np.float64(8.405625775303243), np.float64
     (8.405630330312386)]]
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `FLD`.

3.9 Busy periods

A busy period of order n runs from the instant an arrival raises the number of jobs held by the target to n until it falls back below n. The target may be a station, a station-class pair, or a declared set of stations, and the order n ranges over $1..K$. The analytical reference is the busy period formula of Daduna, exposed by the normalizing constant solver; the simulator measures the same quantity from the sample path.

3.9.1 `busyp_subnetwork`

Mean busy periods of order n for a station, for a station-class pair and for a declared subnetwork of stations.

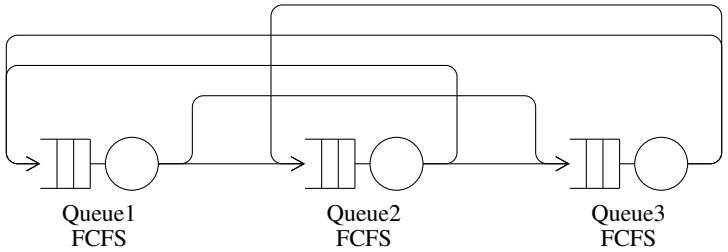

Figure 3.44: Topology of `busyp_subnetwork`: a closed network over 3 queueing stations.

The example is built and solved as follows.

Nodes	3: Queue × 3
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Queue1 – Class1: Exp(1.5) Queue2 – Class1: Exp(0.9) Queue3 – Class1: Exp(2)
Solvers	LDES

Table 3.45: Features of `busyp_subnetwork`.

```
Busy Period of a Subnetwork
The busy period of order n for a set of stations is the time from the instant a
job entering the set finds n-1 jobs in it up to the next instant when fewer than
n remain. SolverNC evaluates it exactly from the normalizing constants of the
subnetwork and of its complement (H. Daduna, "Busy Periods for Subnetworks in
Stochastic Networks: Mean Value Analysis", J. ACM 35(3), 1988); SolverLDES
measures the same quantity along a simulated sample path.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
print('=== Busy Period of a Subnetwork ===\n')
N = 5
model = Network('busyPeriodModel')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS)
jobclass = ClosedClass(model, 'Class1', N, queue1, 0)
queue1.setService(jobclass, Exp(1.5))
queue2.setService(jobclass, Exp(0.9))
queue3.setService(jobclass, Exp(2.0))
P = model.initRoutingMatrix()
P.set(jobclass, jobclass, queue1, queue2, 0.6)
P.set(jobclass, jobclass, queue1, queue3, 0.4)
P.set(jobclass, jobclass, queue2, queue1, 0.7)
P.set(jobclass, jobclass, queue2, queue3, 0.3)
P.set(jobclass, jobclass, queue3, queue1, 0.5)
P.set(jobclass, jobclass, queue3, queue2, 0.5)

    ... (14 source lines omitted; full implementation: python/examples/advanced/)

ldes = SolverLDES(model, opts)
print('\nMean busy period of order n (measured, SolverLDES)')
print(f"{'subnetwork':<18}{'n=1':>10}{'n=3':>10}{'n=5':>10}")
for subnet in subnets:
    b, _ = ldes.getAvgBusyPeriod(subnet, -1, orders)
    print(f'{str(subnet):<18}{b[0]:>10.4f}{b[1]:>10.4f}{b[2]:>10.4f}')
```

Running this edition prints

```
=== Busy Period of a Subnetwork ===

Mean busy period of order n (exact, SolverNC)
subnetwork               n=1       n=3       n=5
[0]                    1.5803    1.2492    0.6667
[1]                   11.9318    4.1381    1.1111
[2]                    0.7101    0.6567    0.5000
[0, 1]               502.3221   25.5284    1.5979

Mean busy period of order n (measured, SolverLDES)
subnetwork               n=1       n=3       n=5
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 2000000 --seed 23000 --busyperiod 5
[0]                    1.5782    1.2547    0.6628
[1]                   11.9004    4.1368    1.1112
[2]                    0.7107    0.6642    0.5012
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 2000000 --seed 23000 --busyperiod 5 --busyperiod-
    subnet 0,1
[0, 1]               487.3380   25.5168    1.5978
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `LDES`.

3.10 Stochastic network calculus

Every other family in these examples reports a mean. The stochastic network calculus reports a *tail*: given a violation probability ε it returns a delay or a queue length that is exceeded with probability at most ε, under any work-conserving scheduling policy at the station. Each flow is carried by a moment-generating-function envelope and each server by the counting process of its service; the min-plus operations compose them across a feed-forward network, at the cost of the burstiness a server adds to the flow it forwards. The bound on the mean that follows by integration is loose, and the decay rate of the tail is asymptotically exact, so the quantile is what these examples read.

3.10.1 `snc_delay_quantile`

A delay quantile with a certified violation probability, and how it tightens deep in the tail.

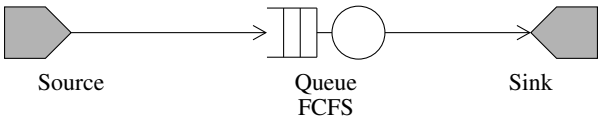

Figure 3.45: Topology of `snc_delay_quantile`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(0.6)
Service	Queue – Class1: Exp(1)

Table 3.46: Features of `snc_delay_quantile`.

The example is built and solved as follows.

```
Stochastic network calculus: a delay quantile with a certified violation
probability.
Every other solver in LINE answers with a MEAN. The 'snc' family of SolverBA
answers with a TAIL: given a violation probability eps, it returns a delay d for
which P{D > d} <= eps holds, and the guarantee is valid for any work-conserving
scheduling policy at the station. That is the quantity a service-level objective
is written against.
The model is a single M/M/1 station, whose exact tail is known in closed form, so
every number below can be checked. Twin of
matlab/examples/advanced/networkCalculus/snc_delay_quantile.m.
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
                                    snc_srv_exp)
print('=== Stochastic network calculus: delay quantile ===\n')
lam = 0.6
mu = 1.0
model = Network('SncQuantile')
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
jobclass = OpenClass(model, 'Class1')
source.setArrival(jobclass, Exp(lam))
queue.setService(jobclass, Exp(mu))
model.link(Network.serialRouting(source, queue, sink))
solver = SolverBA(model, 'snc.upper')
# getPercTable is to the snc family what getAvgTable is to a mean solver: one row

    ... (42 source lines omitted; full implementation: python/examples/advanced/)

print('     the optimal theta approaches log(mu/lam) = %.4f, which is'
```

```
      % math.log(mu / lam))
print('     what makes the backlog decay rate exact')
eps_at, theta_at = snc_bound_delay(arv, srv, d)
print('api: P{D > %.4f} <= %.3e  (theta = %.4f), exact tail %.3e'
      % (d, eps_at, theta_at, math.exp(-(mu - lam) * d)))
```

Running this edition prints

```
=== Stochastic network calculus: delay quantile ===

  Station JobClass  RespTPerc  QLenPerc
0   Queue   Class1   29.60819  23.64494

eps         d bound    d exact    ratio      n bound    n exact    ratio
1e-02       23.3454    11.5129    2.028      18.7266     8.0152    2.336
1e-03       29.6082    17.2694    1.714      23.6449    12.5227    1.888
1e-06       47.9527    34.5388    1.388      38.0313    26.0455    1.460
1e-09       65.9597    51.8082    1.273      52.1418    39.5682    1.318
1e-12       83.7930    69.0776    1.213      66.1120    53.0909    1.245

Station  JobClass    QLen  Util   RespT  ResidT  ArvR  Tput
Source   Class1         0     0       0       0     0   0.6
Queue    Class1    7.9231   0.6  13.205  13.205   0.6   0.6

exact M/M/1: R = 2.5000, Q = 1.5000
snc.upper  : R = 13.2051 (5.3x), Q = 7.9231 (5.3x)

api: d(1e-3) = 29.6082 at theta = 0.4632
     the optimal theta approaches log(mu/lam) = 0.5108, which is
     what makes the backlog decay rate exact
api: P{D > 29.6082} <= 1.000e-03  (theta = 0.4632), exact tail 7.187e-06
```

3.10.2 `snc_tandem_multiclass`

Envelope propagation across a feed-forward tandem, pay-bursts-only-once, blind multiplexing, and the model classes the family refuses.

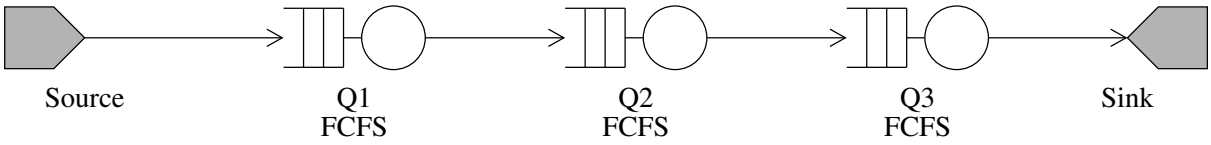

Figure 3.46: Topology of `snc_tandem_multiclass`: an open network over 3 queueing stations.

Nodes	5: Source, Queue × 3, Sink
Classes	1: Class1 (open)
Scheduling	Q1: FCFS; Q2: FCFS; Q3: FCFS
Arrivals	Source – Class1: Exp(0.6)
Service	Q1 – Class1: Exp(1.5) Q2 – Class1: Exp(1.2) Q3 – Class1: Exp(1)

Table 3.47: Features of `snc_tandem_multiclass`.

The example is built and solved as follows.

```
Stochastic network calculus on a feed-forward network: envelope propagation,
blind multiplexing, and the model classes the family refuses.
The bound of snc_delay_quantile.py is a single station. This example is what the
network calculus adds on top of it: a departure envelope carries a flow to the
next hop, cross traffic is subtracted from a shared server, and both operations
cost burstiness, which is visible as the bound loosening downstream and under
sharing.
The reference throughout is the exact Jackson-network solution, since every
station here is an M/M/1 with Poisson input by Burke's theorem. Twin of
matlab/examples/advanced/networkCalculus/snc_tandem_multiclass.m.
Copyright (c) 2012-2026, Imperial College London
```

```
All rights reserved.
                                snc_srv_exp)
print('=== Stochastic network calculus: tandem and shared server ===\n')
lam = 0.6
rates = [1.5, 1.2, 1.0]          # service rates, the last is the bottleneck
model = Network('SncTandem')
source = Source(model, 'Source')
q1 = Queue(model, 'Q1', SchedStrategy.FCFS)
q2 = Queue(model, 'Q2', SchedStrategy.FCFS)
q3 = Queue(model, 'Q3', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
jobclass = OpenClass(model, 'Class1')
source.setArrival(jobclass, Exp(lam))
q1.setService(jobclass, Exp(rates[0]))
q2.setService(jobclass, Exp(rates[1]))

    ... (110 source lines omitted; full implementation: python/examples/advanced/)

    m.link(P)
    SolverBA(m, 'snc.upper').getAvgTable()
print('\nrefusals:')
show_refusal('closed network', closed_model)
show_refusal('probabilistic split downstream of the Source', split_model)
show_refusal('unequal service rates at a shared station', unequal_model)
```

Running this edition prints

```
=== Stochastic network calculus: tandem and shared server ===

Station  JobClass    QLen  Util   RespT  ResidT  ArvR  Tput
Source   Class1         0     0       0       0     0   0.6
Q1       Class1    2.3764   0.4  3.9606  3.9606   0.6   0.6
Q2       Class1    6.4299   0.5  10.716  10.716   0.6   0.6
Q3       Class1    14.843   0.6  24.738  24.738   0.6   0.6

station     R bound    R exact    ratio
Q1           3.9606     1.1111     3.56
Q2          10.7164     1.6667     6.43
Q3          24.7385     2.5000     9.90

end-to-end delay quantile at eps=1e-3
  concatenated (snc_conv) :  42.5195  at theta = 0.4793
  summed per hop          :  65.5128

... (9 captured-output lines omitted; rerun the source example for the full output)

0  Shared   ClassA    55.9018  23.64494
1  Shared   ClassB    55.9018  23.64494
exact per-class response time (aggregate M/M/1, lam=0.6, mu=1): 2.5000

refusals:
  closed network:                              Method 'snc.upper' supports fully open networks only (no
     closed classes).
  probabilistic split downstream of the Source: Method 'snc.upper' requires deterministic routing
     downstream of the Source: station 2 class 1 routes onward with probability 0.5.
  unequal service rates at a shared station:    Method 'snc.upper' requires the classes sharing a station
     to have equal service rates: station 2 carries rates in [1, 2].
```

3.11 Random environments

A random environment is a Markov process over stages, with one model per stage. The environment moves between stages according to a transition process, and the network parameters change with it; the metrics reported are the environment average, weighted by the stage probabilities, but the transient handoff between stages matters when the stage sojourn is comparable with the relaxation time of the network.

Server breakdown and repair is the smallest instance: two stages, one with the server up and one with it down. The case studies at the end of the family are larger, and are the models on which the environment solver is validated.

3.11.1 example_mapqn2renv

Converting a queueing network with MMPP(2) service into an equivalent random environment model.

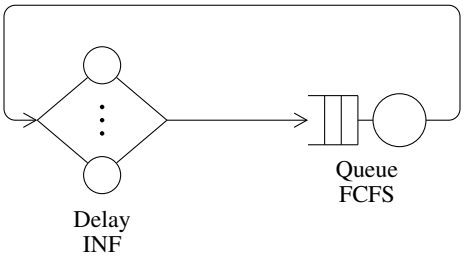

Figure 3.47: Topology of example_mapqn2renv: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue: FCFS
Service	Delay – Class1: Exp(1) Queue – Class1: MMPP2
Solvers	JMT

Table 3.48: Features of example_mapqn2renv.

The example is built and solved as follows.

```
                    GlobalConstants, JMT, MMPP2, MVA, Network, Queue,
                    SchedStrategy, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.SILENT)

# Create closed queueing network with MMPP2 service
model = Network('MMPP_ClosedQN')

delay = Delay(model, 'Delay')                          # Think time station
queue = Queue(model, 'Queue', SchedStrategy.FCFS)     # Service station with MMPP2

N = 5                                                  # Number of jobs
jobclass = ClosedClass(model, 'Class1', N, delay)
delay.setService(jobclass, Exp(1.0))

# Parameters: lambda0, lambda1, sigma01, sigma10
#   Phase 0: slow service (rate 1.0)  -> high queue length
#   Phase 1: fast service (rate 10.0) -> low queue length
#   D0 = [[-1.2, 0.2], [0.3, -10.3]],  D1 = [[1.0, 0], [0, 10.0]]
queue.setService(jobclass, MMPP2(1.0, 10.0, 0.2, 0.3))

model.link(Network.serialRouting(delay, queue))

print('=== MMPP2 Closed QN to Random Environment Transformation ===\n')
print('Original Network:')
print('  Topology: Delay -> MMPP2 Queue -> Delay (cyclic)')
print('  Population: N = %d jobs' % N)

    ... (73 source lines omitted; full implementation: python/examples/advanced/)

            print('\n  Stage %s:' % name)
            print(MVA(stage_model).getAvgTable())
except Exception as err:
    print('Stage solver error: %s' % err)

print('\n=== Transformation Complete ===')
```

Running this edition prints

```
=== MMPP2 Closed QN to Random Environment Transformation ===
```

```
Original Network:
  Topology: Delay -> MMPP2 Queue -> Delay (cyclic)
  Population: N = 5 jobs
  Think time: Exp(1.0)
  Service: MMPP2 with phases 0,1

Transforming to random environment model...
Transformation complete!

Environment Model Structure:
  Environment: MAPQN_Env with 2 stages
  Stage 0 (Phase 0): Closed QN with Exp service rate 1.0 (SLOW)
  Stage 1 (Phase 1): Closed QN with Exp service rate 10.0 (FAST)
  Transitions:

... (42 captured-output lines omitted; rerun the source example for the full output)

Queue    Class1     4.0031  0.99693  4.0154  4.0154  0.99693  0.99693

  Stage Phase_2:
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1     4.3605   4.3605        1        1  4.3605  4.3605
Queue    Class1    0.63952  0.43605  0.14666  0.14666  4.3605  4.3605

=== Transformation Complete ===
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `JMT`.

3.11.2 `renv_basic`

The smallest random environment: two stages, one model per stage, and the stage transition process. No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	FLD, MVA

Table 3.49: Features of `renv_basic`.

The example is built and solved as follows.

```
# Block 1: Create base network model
# First, we define a closed queueing network with a delay and a queue
base_model = Network('BaseModel')
delay = Delay(base_model, 'ThinkTime')
queue = Queue(base_model, 'Fast/Slow Server', SchedStrategy.FCFS)
# Closed class with 5 jobs
N = 5
jobclass = ClosedClass(base_model, 'Jobs', N, delay)
delay.setService(jobclass, Exp(1.0))  # Think time = 1.0
queue.setService(jobclass, Exp(2.0))  # Placeholder service rate
# Connect nodes in a cycle
base_model.link(Network.serialRouting(delay, queue))
# Block 2: Create the random environment
# Define two stages: "Fast" and "Slow" with different service rates
env = Environment('ServerModes', 2)
# Stage 0: Fast mode (service rate = 4.0)
fast_model = base_model.copy()
fast_queue = fast_model.getNodeByName('Fast/Slow Server')
fast_queue.setService(fast_model.classes[0], Exp(4.0))
env.addStage(0, 'Fast', 'operational', fast_model)
# Stage 1: Slow mode (service rate = 1.0)
slow_model = base_model.copy()
slow_queue = slow_model.getNodeByName('Fast/Slow Server')
slow_queue.setService(slow_model.classes[0], Exp(1.0))
env.addStage(1, 'Slow', 'degraded', slow_model)
# Define transitions between stages
```

```
    ... (30 source lines omitted; full implementation: python/examples/advanced/)

    print(f'\nStage {e}:')
    mva_solver = MVA(stage_model)
    stage_table = mva_solver.getAvgTable()
    print(stage_table)
return env
main()
```

Running this edition prints

```
Environment stages:
 Stage Name        Type     Model
     0 Fast operational BaseModel
     1 Slow    degraded BaseModel
   Stage  Name         Type      Model
0      0  Fast  operational  BaseModel
1      1  Slow     degraded  BaseModel
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: python; env: python] completed in
      <t>s.

... (20 captured-output lines omitted; rerun the source example for the full output)

ThinkTime         Jobs      3.2037   3.2037        1        1  3.2037  3.2037
Fast/Slow Server  Jobs      1.7963  0.80093  0.56068  0.56068  3.2037  3.2037

Stage 1:
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station           JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
ThinkTime         Jobs      0.99693  0.99693        1       1  0.99693  0.99693
Fast/Slow Server  Jobs       4.0031  0.99693   4.0154  4.0154  0.99693  0.99693
```

The rows repeat for each of the 2 solvers the script runs (FLD, MVA), which is the point of the example: the same model read by methods of different cost and different exactness.

3.11.3 `renv_container_terminal`

A container terminal case study: MAP arrivals, multiserver and load-dependent stations, several environment stages.

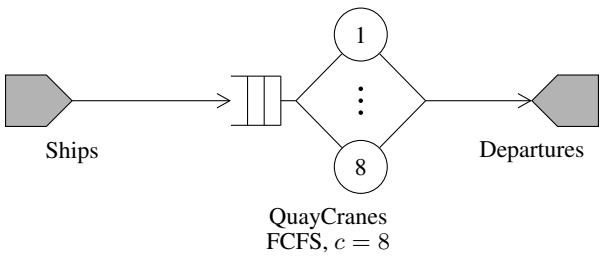

Figure 3.48: Topology of `renv_container_terminal`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Containers (open)
Scheduling	QuayCranes: FCFS ($c = 8$)
Arrivals	Ships – Containers: MAP
Service	QuayCranes – Containers: Exp(16)

Table 3.50: Features of `renv_container_terminal`.

The example is built and solved as follows.

```
               (hour x network-state) random-environment CTMC.
                     MAM, MAP, MVA, Network, OpenClass, Queue,
                     SchedStrategy, Sink, Source, VerboseLevel)
"""One hour-stage network: N carriers cycling yard-Delay -> quay-crane Queue."""
qn = Network('Terminal')
yard = Delay(qn, 'Yard')
cranes = Queue(qn, 'QuayCranes', SchedStrategy.FCFS)
cranes.setNumberOfServers(n_cranes)
containers = ClosedClass(qn, 'Containers', N, yard)
yard.setService(containers, Exp(stage_rate))
cranes.setService(containers, Exp(crane_rate))
qn.link(Network.serialRouting(yard, cranes))
return qn
"""One hour-stage with the internal terminal handling as a closed
load-dependent FES: N containers cycling yard-Delay -> FES, where the FES
serves at rate mu(n) = fes_rate[n-1] when n containers are inside it."""
N = len(fes_rate)
qn = Network('TerminalFES')
yard = Delay(qn, 'Yard')
fesq = Queue(qn, 'TerminalFES', SchedStrategy.PS)
containers = ClosedClass(qn, 'Containers', N, yard)
yard.setService(containers, Exp(stage_rate))
fesq.setService(containers, Exp(fes_rate[0]))            # base rate mu(1)
fesq.setLoadDependence(np.asarray(fes_rate) / fes_rate[0])  # alpha(n) = mu(n)/mu(1)
qn.link(Network.serialRouting(yard, fesq))
return qn

    ... (160 source lines omitted; full implementation: python/examples/advanced/)

open_model.link(Network.serialRouting(ships, quay_open, done))

mean_lambda = float(np.sum(daily_duration_hr * daily_rate_hr) / np.sum(daily_duration_hr))
print('MMPP mean arrival = %.2f/hr, %d cranes @ %.0f/hr, rho = %.3f'
      % (mean_lambda, c_open, crane_rate, mean_lambda / (c_open * crane_rate)))
print(MAM(open_model).getAvgTable())
```

Running this edition prints

```
Rotterdam container terminal: 24 hourly stages, 6 carriers, 2 cranes.

=== Day-averaged quay-crane metrics (container terminal) ===
analyzer           QLen         Util         Tput
exact          3.52580      0.88930     28.45772
statevec       3.52580      0.88930     28.45772
meanfield      3.54037      0.89178     28.53705

QLen error vs exact:  statevec = 2.856e-09 , meanfield = 1.457e-02
Util error vs exact:  statevec = 2.035e-10 , meanfield = 2.479e-03

Day-averaged crane-queue table (state-vector analyzer):
      Station    JobClass    QLen    Util    RespT      Tput
0        Yard  Containers  2.4742  2.4742  0.08694  28.45772
1  QuayCranes  Containers  3.5258  0.8893  0.12390  28.45772


... (19 captured-output lines omitted; rerun the source example for the full output)

  iLB = Minv @ iLB @ iLB
python/line_solver/api/mam/qbd.py:128: RuntimeWarning: invalid value encountered in matmul
  S = S + T @ iLB
```

```
python/line_solver/api/solvers/mam/handler.py:2027: UserWarning: qsys_mapmc failed, falling back: array
     must not contain infs or NaNs
  warnings.warn(f"qsys_mapmc failed, falling back: {e}")
Station     JobClass      QLen     Util     RespT    ResidT    ArvR    Tput
Ships       Containers       0        0         0         0       0   86.77
QuayCranes  Containers  18.515  0.67789  0.21338  0.21338   86.77   86.77
```

3.11.4 renv_fourstages_repairmen

The four-stage version.

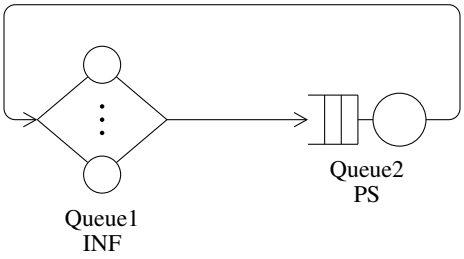

Figure 3.49: Topology of renv_fourstages_repairmen: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 30$)
Scheduling	Queue1: INF; Queue2: PS
Service	Queue1 – Class1: Exp(4) Queue2 – Class1: Exp(1)
Solvers	ENV

Table 3.51: Features of renv_fourstages_repairmen.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
def renv_genqn(rate, N):
    """Helper function to generate a queueing network for random environment advanced."""
    qnet = Network('qn1')
    node = np.empty(2, dtype=object)
    node[0] = Delay(qnet, 'Queue1')
    node[1] = Queue(qnet, 'Queue2', SchedStrategy.PS)
    jobclass = np.empty(1, dtype=object)
    jobclass[0] = ClosedClass(qnet, 'Class1', N, node[0], 0)
    node[0].set_service(jobclass[0], Exp(rate[0]))
    node[1].set_service(jobclass[0], Exp(rate[1]))
    P = qnet.init_routing_matrix()
    P.set(jobclass[0], jobclass[0], [[0, 1], [1, 0]])
    qnet.link(P)
    return qnet
# %%
# Model parameters
N = 30  # Job population
M = 3   # Number of stations
E = 4   # Number of environment stages
# Create environment model
envModel = Environment('MyEnv', E)
envName = ['Stage1', 'Stage2', 'Stage3', 'Stage4']
envType = ['UP', 'DOWN', 'FAST', 'SLOW']

    ... (56 source lines omitted; full implementation: python/examples/advanced/)

                {'iter_max': 100, 'iter_tol': 0.05, 'method': 'default'})
```

```
# Get results
QN, UN, _, TN, _, _ = envSolver.getAvg()
AvgTable = envSolver.getAvgTable()
print("AvgTable =")
print(AvgTable)
```

Running this edition prints

```
Rate matrix:
[[4. 3. 2. 1.]
 [1. 1. 1. 1.]
 [1. 2. 3. 4.]]
Environment transition rates:
[[0.  0.5 0.  0. ]
 [0.  0.  0.5 0.5]
 [0.5 0.  0.  0.5]
 [0.5 0.5 0.  0. ]]
The metasolver considers an environment with 4 stages and a queueing network with 3 stations.
Every time the stage changes, the queueing network will modify the service rates of the stations.
 Stage   Name Type Model
     0 Stage1   UP   qn1
     1 Stage2 DOWN   qn1
     2 Stage3 FAST   qn1
     3 Stage4 SLOW   qn1

... (6 captured-output lines omitted; rerun the source example for the full output)

ENV solver iteration 3
ENV solver iteration 4
ENV solver converged after 4 iterations
ENV analysis [method: default; type: approximate, deterministic; lang: python; env: python] completed in <
    t>s. Iterations: 4.
AvgTable =
  Station JobClass      QLen     Util     RespT     Tput
0  Queue1   Class1   0.43985  0.43985   0.45269  0.97164
1  Queue2   Class1  29.56015  1.00000  29.56015  1.00000
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `ENV`.

3.11.5 `renv_genqn`

A randomly generated network placed in an environment.

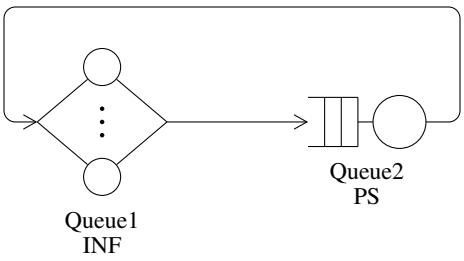

Figure 3.50: Topology of `renv_genqn`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 4$)
Scheduling	Queue1: INF; Queue2: PS
Service	Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(0.5)
Solvers	MVA

Table 3.52: Features of `renv_genqn`.

The example is built and solved as follows.

```
                      Queue, SchedStrategy, VerboseLevel)
"""Build the two-station stage network.
Args:
    rate: the two service rates, (Delay, Queue).
    N: the closed-class population.
Returns:
    The linked Network.
"""
qnet = Network('qn1')
node = np.empty(2, dtype=object)
node[0] = Delay(qnet, 'Queue1')
node[1] = Queue(qnet, 'Queue2', SchedStrategy.PS)
jobclass = np.empty(1, dtype=object)
jobclass[0] = ClosedClass(qnet, 'Class1', N, node[0], 0)
node[0].set_service(jobclass[0], Exp(rate[0]))
node[1].set_service(jobclass[0], Exp(rate[1]))
P = qnet.init_routing_matrix()
P.set(jobclass[0], jobclass[0], [[0, 1], [1, 0]])
qnet.link(P)
return qnet
from line_solver import MVA

GlobalConstants.set_verbose(VerboseLevel.STD)

# one stage on its own: the environment is what varies these rates
qnet = renv_genqn([1.0, 0.5], 4)
print(MVA(qnet).get_avg_table())
```

Running this edition prints

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Class1    0.49921  0.49921       1       1  0.49921  0.49921
Queue2   Class1     3.5008  0.99842  7.0127  7.0127  0.49921  0.49921
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.11.6 `renv_lqn_twostages`

A layered model in a two-stage environment, including its transient.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
m = LayeredNetwork(name)
P1 = Processor(m, 'ClientProcessor', 1, SchedStrategy.PS)
P2 = Processor(m, 'DBProcessor', 1, SchedStrategy.PS)
T1 = Task(m, 'ClientTask', 5, SchedStrategy.REF).on(P1)
T1.set_think_time(Exp.fit_mean(5.0))
T2 = Task(m, 'DBTask', float('inf'), SchedStrategy.INF).on(P2)
E1 = Entry(m, 'ClientEntry').on(T1)
E2 = Entry(m, 'DBEntry').on(T2)
A1 = Activity(m, 'ClientActivity', Exp.fit_mean(1.0)).on(T1)
A1.bound_to(E1).synch_call(E2, 2.5)
A2 = Activity(m, 'DBActivity', Exp.fit_mean(db_mean)).on(T2)
A2.bound_to(E2).replies_to(E2)
return m
# Disabled / off-block cells are None -> NaN (excluded from the totals).
if cell is None:
    return np.nan
if isinstance(cell, dict):
    metric = cell.get('metric')
else:
```

```
    metric = getattr(cell, 'metric', None)
if metric is not None and len(metric) > 0:
    return float(metric[-1])
return np.nan
"""Steady transient aggregate (block-diagonal) Q and T for one LQN stage,
using the same SolverLN layout SolverENV consumes."""
s = SolverLN(model, layer_factory, timespan=[0, T], verbose=False)

    ... (146 source lines omitted; full implementation: python/examples/advanced/)

print("Three-stage (CTMC) ENV throughput = %.4f (bracket [%.4f, %.4f])" % (x3, lo3, hi3))
assert np.isfinite(x3), "Three-stage ENV throughput is not finite"
assert lo3 - tol3 <= x3 <= hi3 + tol3, \
    "Three-stage ENV-averaged throughput is not bracketed"
print("PASS: ENV-over-LQN meanfield ran; throughput bracketed, monotone in P(UP), 3-stage
    bracketed.")
main()
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.11.7 `renv_map_fallback`

An MMPP(2) stage transition process, and the fallback taken when it cannot be represented exactly.

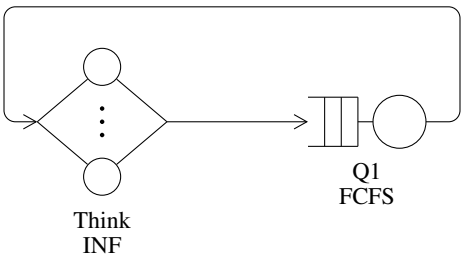

Figure 3.51: Topology of `renv_map_fallback`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: C1 (closed, $N = 5$)
Scheduling	Think: INF; Q1: FCFS
Service	Think – C1: Exp(1) Q1 – C1: MMPP2
Solvers	MVA, ENV, CTMC

Table 3.53: Features of `renv_map_fallback`.

The example is built and solved as follows.

```
                    MVA, Network, Queue, SchedStrategy, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.STD)

# Closed network with an MMPP2 server: MVA and NC have no native MAP support
model = Network('mmppClosed')
delay = Delay(model, 'Think')
queue = Queue(model, 'Q1', SchedStrategy.FCFS)
jobclass = ClosedClass(model, 'C1', 5, delay)
delay.setService(jobclass, Exp(1.0))
queue.setService(jobclass, MMPP2(1.0, 10.0, 0.2, 0.3))  # slow phase 1, fast phase 10
model.link(Network.serialRouting(delay, queue))

# The environment image built for the solver
renv, info = map2renv(model)
print("Random-environment image: %d stages, phase orders %s, MMPP image: %d"
      % (info['nstages'], info['orders'], int(info['is_mmpp'])))
```

```
renv.getStageTable()

# MVA through the fallback, CTMC natively (exact)
print("\nAvgTableMVA")
print(MVA(model).getAvgTable())
print("\nAvgTableCTMC")
print(CTMC(model).getAvgTable())

# The two environment limits, forced
options = MVA.defaultOptions()

    ... (10 source lines omitted; full implementation: python/examples/advanced/)

options = MVA.defaultOptions()
options.config['map_env'] = 'off'
try:
    MVA(model, options).getAvgTable()
except Exception as err:
    print("map_env='off': %s" % err)
```

Running this edition prints

```
Random-environment image: 2 stages, phase orders [2], MMPP image: 1
 Stage    Name Type              Model
     0 Phase_1 item mmppClosed_Phase_1
     1 Phase_2 item mmppClosed_Phase_2

AvgTableMVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
ENV analysis [method: avg; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
      Iterations: 0.
Warning [mapEnvApprox]: This solver has no native support for the non-renewal (MAP/MMPP) processes of this
     model; the reported averages come from its avg random-environment approximation (2 stages, exact-
    modulation image). Set options.config['map_env']='off' to reject the model instead.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Think    C1        3.4427   3.4427        1        1  3.4427  3.4427
Q1       C1        1.5573  0.74842  0.45234  0.45234  3.4427  3.4427

AvgTableCTMC
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

... (13 captured-output lines omitted; rerun the source example for the full output)

AvgTableAvg
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
ENV analysis [method: avg; type: approximate, deterministic; lang: python; env: python] completed in <t>s.
      Iterations: 0.
Warning [mapEnvApprox]: This solver has no native support for the non-renewal (MAP/MMPP) processes of this
     model; the reported averages come from its avg random-environment approximation (2 stages, exact-
    modulation image). Set options.config['map_env']='off' to reject the model instead.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Think    C1        3.4427   3.4427        1        1  3.4427  3.4427
Q1       C1        1.5573  0.74842  0.45234  0.45234  3.4427  3.4427
map_env='off': This model contains features not supported by the solver. Some features are not supported
    by the chosen solver (feature: MMPP2).
```

The rows repeat for each of the 3 solvers the script runs (MVA, ENV, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.11.8 `renv_node_breakdown`

Server breakdown and repair as an environment over a queueing model.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
"""Create a base queueing network model with a single server.
Returns:
    tuple: (model, queue) - the network model and the queue node for use with Env API
"""
model = Network('ServerWithFailures')
# Define nodes
source = Source(model, 'Arrivals')
queue = Queue(model, 'Server', SchedStrategy.FCFS)
sink = Sink(model, 'Departures')
# Define job class
jobclass = OpenClass(model, 'Jobs')
# Set service and arrival rates (UP state)
source.set_arrival(jobclass, Exp(0.8))  # Arrival rate
queue.set_service(jobclass, Exp(2.0))   # Service rate when UP
queue.set_number_of_servers(1)
# Set routing
P = model.init_routing_matrix()
P.add_route(jobclass, source, queue, sink)
model.link(P)
return model, queue
"""Example 1: Using addNodeFailureRepair convenience method with node object (recommended).
    """
print("=" * 70)
print("Example 1: Using addNodeFailureRepair with node object")
print("=" * 70)
model, queue = create_base_model()
# Create environment with 2 stages (UP and DOWN)

    ... (112 source lines omitted; full implementation: python/examples/advanced/)

    print("=" * 70)
except Exception as e:
    print(f"Error running examples: {e}")
    traceback.print_exc()
return build_env_model()
main()
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.11.9 `renv_rotterdam_blending`

The Rotterdam blending case study.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
                      Queue, SchedStrategy, VerboseLevel)
"""Load-dependent FES throughput curve mu(l), l = 0..N, by exact MVA on the
closed subnetwork at each population l (Norton flow-equivalent)."""
mu = np.zeros(N + 1)
for l in range(1, N + 1):
    if which == 'up':
        m = Network('S1')
        eg = Queue(m, 'EntryGates', SchedStrategy.FCFS)
        eg.setNumberOfServers(p['numEntryServers'])
        tv = Delay(m, 'TravelToStack')
        st = [Queue(m, 'Stack%d' % (i + 1), SchedStrategy.FCFS)
              for i in range(p['numStacks'])]
        for q in st:
            q.setNumberOfServers(1)
        cls = ClosedClass(m, 'Trucks', l, eg)
        eg.setService(cls, Exp(1.0 / p['entryServiceTime']))
        tv.setService(cls, Exp(1.0 / p['travelToStackTime']))
        for q in st:
            q.setService(cls, Exp(1.0 / p['stackServiceTime']))
```

```
        ns = 2 + p['numStacks']
        R = np.zeros((ns, ns))
        R[0, 1] = 1.0                                    # EntryGates -> TravelToStack
        R[1, 2:ns] = 1.0 / p['numStacks']                # TravelToStack -> Stack_i
        R[2:ns, 0] = 1.0                                 # Stack_i -> EntryGates
        P = m.initRoutingMatrix()
        P.set(cls, cls, R)

    ... (112 source lines omitted; full implementation: python/examples/advanced/)

    mu2 = fes_curve(N, params, 'down')
    W_bl, Qex_bl = blend_soqn(N, lam, dur_min, frac_w, mu1, mu2, tail_factor)
    j = int(np.flatnonzero(sim_N == N)[0])
    print('%4d | %10.4f %10.4f %6.2f | %10.4f %10.4f %6.2f'
          % (N, W_bl, sim_W[j], 100 * abs(W_bl - sim_W[j]) / sim_W[j],
             Qex_bl, sim_Qex[j], 100 * abs(Qex_bl - sim_Qex[j]) / sim_Qex[j]))
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.11.10 `renv_threestages_repairmen`

The three-stage version.

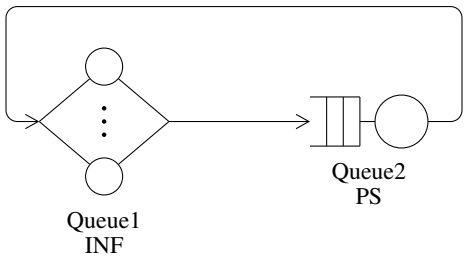

Figure 3.52: Topology of `renv_threestages_repairmen`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 2$)
Scheduling	Queue1: INF; Queue2: PS
Service	Queue1 – Class1: Exp(3) Queue2 – Class1: Exp(1)
Solvers	ENV

Table 3.54: Features of `renv_threestages_repairmen`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
def circul(c):
    """Returns a circulant matrix of order c (MATLAB-compatible).
    Generates a circulant matrix using the same algorithm as MATLAB's circul function.
    For scalar input c, generates a cxc circulant matrix with last element of the
    generating vector set to 1.
    Args:
        c: Either an integer (order) or a vector for the first row
    Returns:
        numpy.ndarray: Circulant matrix matching MATLAB behavior
    """
    if np.isscalar(c):
        if c == 1:
            return np.array([[1]])
```

```
        else:
            v = np.zeros(c)
            v[-1] = 1  # Last element = 1
            return circul(v)
    # c is a vector - use MATLAB's algorithm: C = sum_t c(t) * R^t
    # where R is the right circular shift matrix
    n = len(c)
    I = np.eye(n)
    R = I[:, list(range(1, n)) + [0]]  # MATLAB: I(:, [2:n, 1])
    C = np.zeros((n, n))

    ... (89 source lines omitted; full implementation: python/examples/advanced/)

for i, gen in enumerate(stageInfGen):
    print(f"    {{{gen.shape[0]}x{gen.shape[1]} double}}", end="")
print()
# Get environment-averaged performance metrics
avg_table = envSolver.getAvgTable()
print(avg_table)
```

Running this edition prints

```
Rate matrix:
[[3. 2. 1.]
 [1. 2. 3.]]
Environment transition rates (circulant matrix):
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
infGen =
  36x36 sparse double matrix (120 nonzeros)
   (1,1)             -4
   (1,2)              3
   (1,4)              1
   (2,2)             -4
   (2,3)              3
   (2,5)              1
   (3,3)             -4

... (28 captured-output lines omitted; rerun the source example for the full output)

ENV solver iteration 1
ENV solver iteration 2
ENV solver iteration 3
ENV solver converged after 3 iterations
ENV analysis [method: default; type: approximate, deterministic; lang: python; env: python] completed in <
    t>s. Iterations: 3.
  Station JobClass     QLen     Util    RespT     Tput
0  Queue1   Class1  0.83066  0.83066  0.62301  1.33332
1  Queue2   Class1  1.16934  0.75817  0.85676  1.36484
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `ENV`.

3.11.11 `renv_twostages_repairmen`

A repairman model in a two-stage random environment.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 1$)
Scheduling	Queue1: INF; Queue2: PS
Service	Queue1 – Class1: Exp(2) Queue2 – Class1: Exp(1)
Solvers	ENV

Table 3.55: Features of `renv_twostages_repairmen`.

The example is built and solved as follows.

```
# %%
```

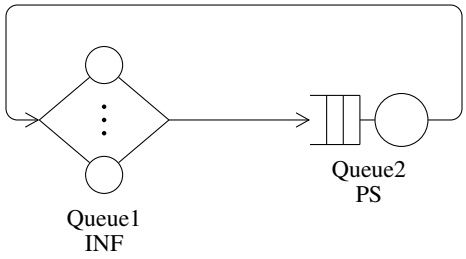

Figure 3.53: Topology of `renv_twostages_repairmen`: a closed network over 2 queueing stations.

```
GlobalConstants.set_verbose(VerboseLevel.STD)
# %%
N = 1
M = 2
E = 2
envModel = Environment('MyEnv', E)
envName = ['Stage1', 'Stage2']
envType = ['UP', 'DOWN']
rate = np.array([[2,1],[1,2]])
# %%
def renv_genqn(rate, N):
    # sn1
    qnet = Network('qn1')
    node = np.empty(2, dtype=object)
    node[0] = Delay(qnet, 'Queue1')
    node[1] = Queue(qnet, 'Queue2', SchedStrategy.PS)
    jobclass = np.empty(1, dtype=object)
    jobclass[0] = ClosedClass(qnet, 'Class1', N, node[0], 0)
    node[0].set_service(jobclass[0], Exp(rate[0]))
    node[1].set_service(jobclass[0], Exp(rate[1]))
    P = qnet.init_routing_matrix()
    P.set(jobclass[0],jobclass[0], [[0,1],[1,0]])
    qnet.link(P)
    return qnet
# %%

    ... (49 source lines omitted; full implementation: python/examples/advanced/)

# scoptions.verbose = False
# envSolver = ENV(envModel, lambda model: CTMC(model, scoptions), options)
# QNc, UNc, TNc = envSolver.avg()
# AvgTableC = envSolver.avg_table()
# print("CTMC Results:")
# print("Average Table (CTMC):", AvgTableC)
```

Running this edition prints

```
Stage Table:
 Stage   Name Type Model
     0 Stage1   UP   qn1
     1 Stage2 DOWN   qn1
ENV solver iteration 1
ENV solver iteration 2
ENV solver iteration 3
ENV solver iteration 4
ENV solver iteration 5
ENV solver converged after 5 iterations
ENV analysis [method: default; type: approximate, deterministic; lang: python; env: python] completed in <
    t>s. Iterations: 5.
Average queue lengths (QN): [[0.5555361]
 [0.4444639]]
Average utilizations (UN): [[0.5555361]
 [0.4444639]]
Average throughputs (TN): [[0.69128289]
 [0.69134125]]

Average Table:
```

```
   Station JobClass     QLen     Util    RespT     Tput
0  Queue1   Class1  0.55554  0.55554  0.80363  0.69128
1  Queue2   Class1  0.44446  0.44446  0.64290  0.69134
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `ENV`.

3.12 Layered cache-queueing models

A `CacheTask` inside a layered model exposes `ItemEntry` points, and a call to one is a cache request whose hit or miss outcome selects the downstream branch. The replacement policy is simulated directly, so hit and miss are exact sample-path events rather than a hit-ratio parameter.

3.12.1 `lcq_singlehost`

A cache task and its item entries inside a layered model on one processor.

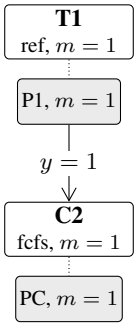

Figure 3.54: Call graph of `lcq_singlehost`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (ps, $m = 1$), PC (ps, $m = 1$)
Tasks	2: T1 (ref, $m = 1$), C2 (fcfs, $m = 1$)
Entries	2: E1, I2
Activities	4, host demands A1: 0.0, AC2: 0.0, AC2h: 1.0, AC2m: 2.0
Calls	1 (1 synchronous, 0 asynchronous)

Table 3.56: Features of `lcq_singlehost`.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# %%
model = LayeredNetwork('cacheInLayeredNetwork')
# Client
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
# Cache task
totalitems = 4
cachecapacity = 2
# Create uniform access probabilities using Python list
access_probs = [1.0 / totalitems] * totalitems
pAccess = DiscreteSampler(access_probs)
PC = Processor(model, 'PC', 1, SchedStrategy.PS)
C2 = CacheTask(model, 'C2', totalitems, cachecapacity, ReplacementStrategy.RR, 1).on(PC)
I2 = ItemEntry(model, 'I2', totalitems, pAccess).on(C2)
```

```
# Definition of activities
A1 = Activity(model, 'A1', Immediate()).on(T1).bound_to(E1).synch_call(I2, 1)
AC2 = Activity(model, 'AC2', Immediate()).on(C2).bound_to(I2)
AC2h = Activity(model, 'AC2h', Exp(1.0)).on(C2).replies_to(I2)  # Cache hit
AC2m = Activity(model, 'AC2m', Exp(0.5)).on(C2).replies_to(I2)  # Cache miss
# Add cache access precedence
C2.add_precedence(ActivityPrecedence.cache_access(AC2, [AC2h, AC2m]))
# %%
lnoptions = LN.default_options()
# lnoptions.iter_max = 1  # Note: iter_max option may not be available
lnoptions.verbose = True
options = MVA.default_options()
options.verbose = False
solver = LN(model, lambda model: MVA(model, options), lnoptions)
AvgTable = solver.avg_table()
print(AvgTable)
```

Running this edition prints

```
Warning [SolverMVA]: SolverMVA does not handle immediate feedback (immfeed); the solver will treat self-
     loops as class-switching with re-queueing.
[SolverMVA] Message cast more than once, repetitions will not be printed for 60 seconds.
Node  NodeType      QLen     Util  RespT  ResidT  ArvR     Tput
P1    Processor      NaN        0    NaN     NaN   NaN      NaN
PC    Processor      NaN        1    NaN     NaN   NaN      NaN
T1    RefTask          1        0    NaN       0   NaN  0.66667
C2    Task             1        1    NaN     1.5   NaN  0.66667
E1    Entry            1        0    1.5     NaN   NaN  0.66667
I2    Entry            1        1    1.5     NaN   NaN  0.66667
A1    Activity         1        0    1.5       0   NaN  0.66667
AC2   Activity         0        0      0       0   NaN        0
AC2h  Activity   0.33333  0.33333      1     0.5   NaN  0.33333
AC2m  Activity   0.66667  0.66667      2       1   NaN  0.33333
```

3.12.2 `lcq_threehosts`

The three-processor version, where the cache and its clients are on separate hosts.

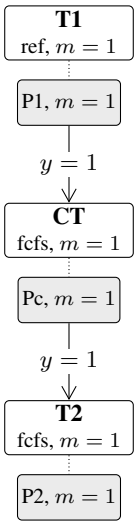

Figure 3.55: Call graph of `lcq_threehosts`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
# %%
GlobalConstants.set_verbose(VerboseLevel.SILENT)
# %%
```

Processors	3: P1 (ps, $m = 1$), Pc (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	3: T1 (ref, $m = 1$), CT (fcfs, $m = 1$), T2 (fcfs, $m = 1$)
Entries	3: E1, IE, E2
Activities	5, host demands A1: 0.0, Ac: 0.0, Ac_hit: 1.0, Ac_miss: 2.0, A2: 0.2
Calls	2 (2 synchronous, 0 asynchronous)

Table 3.57: Features of `lcq_threehosts`.

```
# Example of Layered Network with a multi-level cache
model = LayeredNetwork('LQNwithCaching')
nusers = 1
ntokens = 1
# Client
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', nusers, SchedStrategy.REF).on(P1)
E1 = Entry(model, 'E1').on(T1)
# Cache task
totalitems = 4
cachecapacity = [1, 1]
pAccess = DiscreteSampler([1.0/totalitems] * totalitems)
PC = Processor(model, 'Pc', 1, SchedStrategy.PS)
C2 = CacheTask(model, 'CT', totalitems, cachecapacity, ReplacementStrategy.RR, ntokens).on(PC
    )
I2 = ItemEntry(model, 'IE', totalitems, pAccess).on(C2)
# Server
P3 = Processor(model, 'P2', 1, SchedStrategy.PS)
T3 = Task(model, 'T2', 1, SchedStrategy.FCFS).on(P3)
E3 = Entry(model, 'E2').on(T3)
A3 = Activity(model, 'A2', Exp(5.0)).on(T3).bound_to(E3).replies_to(E3)
# Definition of activities
A1 = Activity(model, 'A1', Immediate()).on(T1).bound_to(E1).synch_call(I2, 1)
AC2 = Activity(model, 'Ac', Immediate()).on(C2).bound_to(I2)

    ... (11 source lines omitted; full implementation: python/examples/advanced/)

# %%
options2 = MVA.default_options()
options2.verbose = 0
solver2 = LN(model, lambda m: MVA(m, options2), lnoptions)
AvgTable2 = solver2.avg_table()
print(AvgTable2)
```

Running this edition prints

```
Warning [SolverNC]: SolverNC does not handle immediate feedback (immfeed); the solver will treat self-
    loops as class-switching with re-queueing.
[SolverNC] Message cast more than once, repetitions will not be printed for 60 seconds.
Warning [SolverNC]: SolverNC does not handle immediate feedback (immfeed); the solver will treat self-
    loops as class-switching with re-queueing.
[SolverNC] Message cast more than once, repetitions will not be printed for 60 seconds.
Node     NodeType      QLen     Util    RespT    ResidT  ArvR     Tput
P1       Processor      NaN        0      NaN       NaN   NaN      NaN
Pc       Processor      NaN  0.93621      NaN       NaN   NaN      NaN
P2       Processor      NaN   0.0625      NaN       NaN   NaN      NaN
T1       RefTask          1        0      NaN         0   NaN  0.61855
CT       Task       0.99794  0.93621      NaN    1.5167   NaN  0.61727
T2       Task        0.0625   0.0625      NaN       0.2   NaN   0.3125
E1       Entry            1        0   1.6167       NaN   NaN  0.61855
IE       Entry      0.99794  0.93621   1.6167       NaN   NaN  0.61727
E2       Entry       0.0625   0.0625      0.2       NaN   NaN   0.3125
A2       Activity    0.0625   0.0625      0.2       0.2   NaN   0.3125
A1       Activity         1        0   1.6167         0   NaN  0.61855

... (12 captured-output lines omitted; rerun the source example for the full output)

E1       Entry            1        0      1.6       NaN   NaN    0.625
IE       Entry            1   0.9375      1.6       NaN   NaN    0.625
E2       Entry       0.0625   0.0625      0.2       NaN   NaN   0.3125
A2       Activity    0.0625   0.0625      0.2       0.2   NaN   0.3125
A1       Activity         1        0      1.6         0   NaN    0.625
Ac       Activity         0        0        0         0   NaN        0
```

```
Ac_hit   Activity   0.3125  0.3125      1     0.5   NaN  0.3125
Ac_miss  Activity   0.6875   0.625    2.2       1   NaN  0.3125
```

3.13 Pass and swap

Under the pass-and-swap discipline a job that finds an incompatible job in service swaps with it rather than waiting, subject to a compatibility relation between classes. The initial placement of a closed population is therefore part of the model: the state must be set in the order the discipline assumes, and a placement that ignores it changes the answer.

3.13.1 `pas_closed_tandem_fig6`

The closed tandem of the reference paper, figure 6.

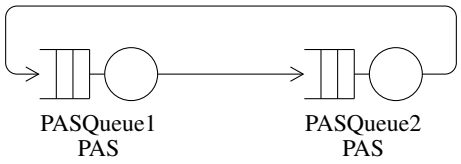

Figure 3.56: Topology of `pas_closed_tandem_fig6`: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	6: Class1 (closed, $N = 1$), Class2 (closed, $N = 1$), Class3 (closed, $N = 1$), Class4 (closed, $N = 1$), Class5 (closed, $N = 1$), Class6 (closed, $N = 1$)
Scheduling	PASQueue1: PAS; PASQueue2: PAS
Service	PASQueue1 – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1); Class4: Exp(1); Class5: Exp(1); Class6: Exp(1) PASQueue2 – Class1: Exp(1.3); Class2: Exp(1.3); Class3: Exp(1.3); Class4: Exp(1.3); Class5: Exp(1.3); Class6: Exp(1.3)
Features	finite buffer 6 at PASQueue1; finite buffer 6 at PASQueue2; class switching on 2 links
Solvers	CTMC

Table 3.58: Features of `pas_closed_tandem_fig6`.

The example is built and solved as follows.

```
Closed tandem of two pass-and-swap (PAS) queues, reproducing Figures 5 and 6
of Comte & Dorsman, "Pass-and-Swap Queues" (2021, arXiv:2009.12299).
    Topology (Fig. 6):   --> PASQueue1 --> PASQueue2 -->   (closed tandem)
Six classes, one customer each; both queues share the Figure 5 swapping graph
(edges 1-3, 1-4, 2-4, 2-5, 3-6, 4-6, 5-6). Head-only single-server service;
the departing customer is chosen by the pass-and-swap scan and routed to the
back of the other queue.
A closed PAS network with a non-empty swapping graph is REDUCIBLE (the
pass-and-swap mechanism conserves the placement order), so the initial job
placement is a REQUIRED model input that selects the recurrent component on
which the stationary distribution is supported. We use the placement of
Fig. 6a: queue 1 = (1,2,3,4,5,6), queue 2 empty. The PAS state list is 1-based
(jobClass+1, 0 = empty), so setState uses 1..6.
The CTMC result is validated against an independent product-form reference that
builds the Markov chain directly from the pass-and-swap algorithm.
MU1, MU2 = 1.0, 1.3
EDGES = [(1, 3), (1, 4), (2, 4), (2, 5), (3, 6), (4, 6), (5, 6)]
G = np.zeros((6, 6))
for a, b in EDGES:
    G[a - 1, b - 1] = 1
    G[b - 1, a - 1] = 1
model = Network('PASclosedTandem')
q1 = Queue(model, 'PASQueue1', SchedStrategy.PAS)
```

```
q2 = Queue(model, 'PASQueue2', SchedStrategy.PAS)
jobclass = [ClosedClass(model, 'Class%d' % (r + 1), 1, q1) for r in range(6)]
q1.setService(lambda c: MU1)        # head-only: mu(prefix)=mu1

    ... (81 source lines omitted; full implementation: python/examples/advanced/)

for i in range(12):
    print('  %-9s  Class%-2d %9.5f %9.5f' % (names[i], (i % 6) + 1, Qref[i], qc[i]))
err = np.max(np.abs(qc - Qref))
print('\nCTMC vs reference: max|dQ| = %.3e' % err)
assert err <= 1e-6, 'CTMC does not match the pass-and-swap reference'
print('PASS: CTMC matches the pass-and-swap reference.')
```

Running this edition prints

```
Warning [solver_ctmc]: The generator has more than one closed communicating class (4956 of 5040 states
    cannot reach the one containing the initial state), so its stationary distribution is not unique.
    Results are those of the class the initial state selects, and depend on it: an order-preserving
    discipline under routing that admits no overtaking freezes the relative service order at time zero,
    so declaring the classes in a different order gives a different, equally valid answer. Product-form
    results (MVA, NC) normalize over the whole space instead and will not agree.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station    JobClass     QLen     Util    RespT  ResidT      ArvR      Tput
PASQueue1  Class1    0.89815  0.47156  10.273  10.273  0.087429  0.087429
PASQueue1  Class2    0.89815  0.47156  10.273  10.273  0.087429  0.087429
PASQueue1  Class3    0.64364        0  3.9298  3.9298   0.16378   0.16378
PASQueue1  Class4    0.63563        0  3.8248  3.8248   0.16619   0.16619
PASQueue1  Class5    0.64364        0  3.9298  3.9298   0.16378   0.16378
PASQueue1  Class6    0.27452        0       1       1   0.27452   0.27452
PASQueue2  Class1    0.10185        0   1.165   1.165  0.087429  0.087429
PASQueue2  Class2    0.10185        0   1.165   1.165  0.087429  0.087429
PASQueue2  Class3    0.35636        0  2.1758  2.1758   0.16378   0.16378
PASQueue2  Class4    0.36437        0  2.1926  2.1926   0.16619   0.16619
PASQueue2  Class5    0.35636        0  2.1758  2.1758   0.16378   0.16378
PASQueue2  Class6    0.72548  0.72548  2.6427  2.6427   0.27452   0.27452

... (9 captured-output lines omitted; rerun the source example for the full output)

  PASQueue2  Class2    0.10185   0.10185
  PASQueue2  Class3    0.35636   0.35636
  PASQueue2  Class4    0.36437   0.36437
  PASQueue2  Class5    0.35636   0.35636
  PASQueue2  Class6    0.72548   0.72548

CTMC vs reference: max|dQ| = 1.887e-15
PASS: CTMC matches the pass-and-swap reference.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.13.2 `pas_compatibility_5class`

Five classes and the compatibility matrix that says which may swap with which.

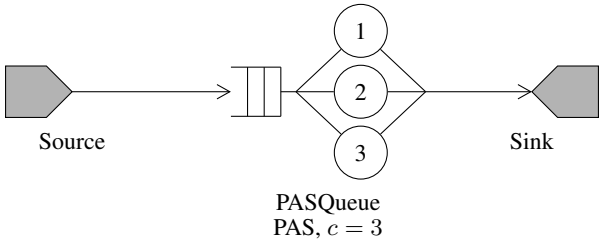

Figure 3.57: Topology of `pas_compatibility_5class`: an open network over 1 queueing station.

The example is built and solved as follows.

Nodes	3: Source, Queue, Sink
Classes	5: Class1 (open), Class2 (open), Class3 (open), Class4 (open), Class5 (open)
Scheduling	PASQueue: PAS ($c = 3$)
Arrivals	Source – Class1: Exp(0.5); Class2: Exp(0.4); Class3: Exp(0.3); Class4: Exp(0.2); Class5: Exp(0.1)
Service	PASQueue – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1); Class4: Exp(2); Class5: Exp(1)
Features	finite buffer 3 at PASQueue
Solvers	CTMC

Table 3.59: Features of `pas_compatibility_5class`.

```
Pass-and-swap (PAS) queue: an order-independent queue with a swapping graph.
Reproduces the five-class, three-server compatibility example of
Dorsman & Gardner (2024), "New directions in pass-and-swap queues",
Queueing Systems 107:205-256 (Figs 1-2). A PAS station is parameterized by the
total service-rate function mu(c) of the ordered state vector c (0-based class
indices, c[0] the oldest job) and by a swapping graph G; both are properties of
the Queue object. The stationary distribution is the order-independent product
form and is invariant to G.
# Server compatibility (Fig 1): three unit-rate servers over five classes.
# mu(c) = number of servers compatible with at least one class present in c.
comp = np.array([[1, 0, 0, 1, 0],     # server 1: classes {0,3}
                 [0, 1, 0, 1, 0],     # server 2: classes {1,3}
                 [0, 0, 1, 0, 1]])    # server 3: classes {2,4}
def mu_fun(c):
    c = np.asarray(c, dtype=int)
    return float(np.sum(np.any(comp[:, c], axis=1))) if len(c) else 0.0
model = Network('PAScompatibility')
source = Source(model, 'Source')
queue = Queue(model, 'PASQueue', SchedStrategy.PAS)
sink = Sink(model, 'Sink')
lam = [0.5, 0.4, 0.3, 0.2, 0.1]
classes = [OpenClass(model, f'Class{r+1}') for r in range(5)]
for r in range(5):
    source.setArrival(classes[r], Exp(lam[r]))
# PAS service: specify the rate function mu(c) as a whole (no per-class rates)
queue.setService(mu_fun)

    ... (8 source lines omitted; full implementation: python/examples/advanced/)

P = model.initRoutingMatrix()
for r in range(5):
    P[classes[r]] = Network.serialRouting(source, queue, sink)
model.link(P)
AvgTable = CTMC(model, cutoff=3).getAvgTable()
print(AvgTable)
```

Running this edition prints

```
CTMC solver using state space cutoff = 3 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded by the source SCCs
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass     QLen      Util     RespT   ResidT      ArvR      Tput
Source    Class1          0         0         0        0         0   0.39099
Source    Class2          0         0         0        0         0   0.31279
Source    Class3          0         0         0        0         0   0.23459
Source    Class4          0         0         0        0         0    0.1564
Source    Class5          0         0         0        0         0  0.078198
PASQueue  Class1    0.56149   0.13033    1.4361   1.4361   0.39099   0.39099
PASQueue  Class2    0.42141   0.10426    1.3472   1.3472   0.31279   0.31279
PASQueue  Class3    0.30532  0.078198    1.3015   1.3015   0.23459   0.23459
PASQueue  Class4    0.11235  0.033661   0.71837  0.71837    0.1564    0.1564
PASQueue  Class5    0.10177  0.026066    1.3015   1.3015  0.078198  0.078198
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and

throughput of each station-class pair, as returned by `CTMC`.

3.13.3 `pas_cyclic_ctmc_vs_bruteforce`

A cyclic pass-and-swap network solved by CTMC and by brute-force enumeration, as a cross-check. No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	CTMC, LDES

Table 3.60: Features of `pas_cyclic_ctmc_vs_bruteforce`.

The example is built and solved as follows.

```
Validate the CTMC (and LDES) solvers on a CLOSED CYCLIC network of two
pass-and-swap (PAS / order-independent) queues against the exact
product-form brute-force normalizing constant.
    Topology:   --> Queue1 --> Queue2 -->   (cyclic, R closed classes)
PAS queues are order-independent, so the network is product-form. The
brute force enumerates the whole ORDERED state space to obtain the
normalizing constant G, hence exact per-class throughputs and mean queue
lengths, which are compared against CTMC (exact) and LDES
(simulation, agreement only within Monte-Carlo noise).
Both stations must be valid order-independent (OI) queues: the TOTAL
service rate mu_i(c) must depend only on the customer multiset.
  station 1: M/M/k OI queue, class-independent  -> mu1(c) = min(n,k1)*s1
  station 2: infinite-server, class-dependent   -> mu2(c) = sum_j beta2[c_j]
For station 2 the prefix sums depend on order, so the ordered-state
enumeration is genuinely needed (it does not collapse to class counts).
Class indices passed to mu(.) and the swap graph are 0-based (Pythonic).
K = [2, 2]                  # per-class closed populations
s1, k1 = 1.0, 2             # station 1: 2-server OI (M/M/2)
beta2 = [1.5, 1.0]          # station 2: per-class infinite-server rates
R = len(K)
def mu1(c):
    c = np.asarray(c)
    return float(min(len(c), k1) * s1)
def mu2(c):
    c = np.asarray(c, dtype=int)
    return float(sum(beta2[ci] for ci in c))

    ... (108 source lines omitted; full implementation: python/examples/advanced/)

tol_sim = 0.02
relX = np.max(np.abs(Xbf - Xl) / Xbf)
relQ = np.max(np.abs(Qbf - Ql) / Qbf)
print("\nLDES:  max rel|dX| = %.2f%%   max rel|dQ| = %.2f%%" % (100 * relX, 100 * relQ))
assert relX <= tol_sim and relQ <= tol_sim, "LDES vs brute-force exceeds 2%"
print("PASS: LDES matches brute-force within %.0f%% (simulation noise)." % (100 * tol_sim))
```

Running this edition prints

```
Brute-force normalizing constant G = 3.69444444444

  CTMC has 1 SCCs; absorption seeded by the source SCCs
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station    JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
PASQueue1  Class1     1.3534  0.48496   1.3953   1.3953  0.96992  0.96992
PASQueue1  Class2     1.1805  0.40977   1.4404   1.4404  0.81955  0.81955
PASQueue2  Class1    0.64662  0.16165  0.66667  0.66667  0.96992  0.96992
PASQueue2  Class2    0.81955  0.20489        1        1  0.81955  0.81955

--- CTMC vs brute-force ---
metric   class       brute-force           CTMC    abs.err
Tput     Class1    0.9699248120   0.9699248120   1.11e-16
Tput     Class2    0.8195488722   0.8195488722   1.11e-16
```

```
QLen q1   Class1    1.3533834586   1.3533834586   2.22e-16
QLen q1   Class2    1.1804511278   1.1804511278   0.00e+00

... (9 captured-output lines omitted; rerun the source example for the full output)

Tput     Class2        0.819549       0.819969     0.05%
QLen q1   Class1        1.353383       1.350350     0.22%
QLen q1   Class2        1.180451       1.179178     0.11%
QLen q2   Class1        0.646617       0.649650     0.47%
QLen q2   Class2        0.819549       0.820822     0.16%

LDES:  max rel|dX| = 0.55%   max rel|dQ| = 0.47%
PASS: LDES matches brute-force within 2% (simulation noise).
```

The rows repeat for each of the 2 solvers the script runs (CTMC, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

3.13.4 pas_cyclic_normconst_bruteforce

The same cross-check on the normalizing constant.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
  w(c1,c2) = Phi_1(c1) * Phi_2(c2),
  Phi_i(c) = prod_{j=1..n} e_{i,c_j} / mu_i(c_1,...,c_j),
  G = sum_{(c1,c2)} Phi_1(c1) * Phi_2(c2)
    = sum_{m<=K} S_1(m) * S_2(K-m),
"""Sum of the OI balance function over every ordered arrangement (distinct
multiset permutation) of the per-class multiset `counts`.
Identical-class jobs yield a single ordered state, so the recursion branches
on the next CLASS placed, generating each distinct sequence exactly once.
"""
return _oi_dfs(list(counts), [], 1.0, e, mu)
if all(c == 0 for c in counts):
    return w                          # completed an ordered state
total = 0.0
for r in range(len(counts)):
    if counts[r] > 0:
        c2 = list(counts)
        c2[r] -= 1
        p2 = prefix + [r]
        total += _oi_dfs(c2, p2, w * e[r] / mu(p2), e, mu)
return total
"""Total OI service rate of the ordered prefix c (0-based class ids) for a
k-server PAS station: the base rates of the first min(n,k) positions."""
if not c:
    return 0.0
return sum(beta[j] for j in c[:min(len(c), k)])
"""Every per-class count vector m with 0 <= m_r <= K_r."""

    ... (56 source lines omitted; full implementation: python/examples/advanced/)

        Km[r] -= 1
        X[r] = norm_const(Km, e1, e2, mu1, mu2) / G
print('Per-class throughput X = [%s]' % ' '.join('%.6g' % x for x in X))
selftest_single_class()
return G
pas_cyclic_normconst_bruteforce()
```

Running this edition prints

```
Closed cyclic PAS network: R=2 classes, population K=[2 2]
Brute-force normalizing constant G = 3.69444444444444
Per-class throughput X = [0.969925 0.819549]
Self-test (single-class Gordon-Newell): PASS (G=0.0855195473251)
```

3.13.5 pas_mmk

The pass-and-swap discipline on an M/M/k station.

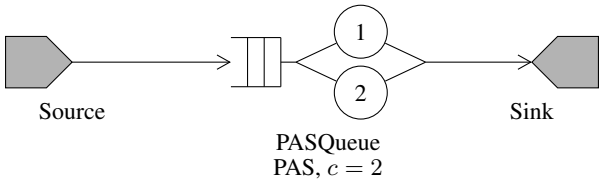

Figure 3.58: Topology of pas_mmk: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	PASQueue: PAS ($c = 2$)
Arrivals	Source – Class1: Exp(0.7); Class2: Exp(0.5)
Service	PASQueue – Class1: Exp(1); Class2: Exp(1)
Features	finite buffer 4 at PASQueue
Solvers	CTMC

Table 3.61: Features of pas_mmk.

The example is built and solved as follows.

```
Pass-and-swap (PAS) station as a multiserver order-independent (M/M/K) queue.
The M/M/K queue satisfies the order-independence conditions (Dorsman & Gardner
2024, Sect. 2.2): with K unit-rate servers the total service rate in state c is
mu(c) = min(n, K), where n is the number of jobs. With an empty swapping graph
the PAS station reduces to a plain order-independent queue.
K = 2   # number of servers
def mu_fun(c):
    c = np.asarray(c)
    return float(min(len(c), K))
model = Network('PASmmk')
source = Source(model, 'Source')
queue = Queue(model, 'PASQueue', SchedStrategy.PAS)
sink = Sink(model, 'Sink')
lam = [0.7, 0.5]
classes = [OpenClass(model, f'Class{r+1}') for r in range(2)]
for r in range(2):
    source.setArrival(classes[r], Exp(lam[r]))
queue.setService(mu_fun)
queue.setSwapGraph(np.zeros((2, 2)))   # empty graph: plain order-independent queue
queue.setNumberOfServers(K)
queue.setCap(4)
P = model.initRoutingMatrix()
for r in range(2):
    P[classes[r]] = Network.serialRouting(source, queue, sink)
model.link(P)
AvgTable = CTMC(model, cutoff=4).getAvgTable()
print(AvgTable)
```

Running this edition prints

```
CTMC solver using state space cutoff = 4 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source    Class1          0        0        0        0        0  0.64976
Source    Class2          0        0        0        0        0  0.46411
PASQueue  Class1    0.80328  0.32488   1.2363   1.2363  0.64976  0.64976
```

```
PASQueue  Class2    0.57377  0.23206  1.2363  1.2363  0.46411  0.46411
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.13.6 pas_nc_sampling

The normalizing constant of a pass-and-swap network, by sampling.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
                    Queue, SchedStrategy, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.SILENT)

MU1, MU2 = 1.0, 1.3
EDGES = [(1, 3), (1, 4), (2, 4), (2, 5), (3, 6), (4, 6), (5, 6)]
G = np.zeros((6, 6))
for a, b in EDGES:
    G[a - 1, b - 1] = 1
    G[b - 1, a - 1] = 1

model = Network('PASsampling')
q1 = Queue(model, 'PASQueue1', SchedStrategy.PAS)
q2 = Queue(model, 'PASQueue2', SchedStrategy.PAS)
jobclass = [ClosedClass(model, 'Class%d' % (r + 1), 1, q1) for r in range(6)]
q1.setService(lambda c: MU1)       # head-only single server
q2.setService(lambda c: MU2)
q1.setSwapGraph(G); q1.setNumberOfServers(1); q1.setCap(6)
q2.setSwapGraph(G); q2.setNumberOfServers(1); q2.setCap(6)
# MATLAB writes P{class}(q1,q2)=1 into the routing matrix and links THAT;
# setProbRouting alone leaves the matrix empty here, and the chain then has
# no arcs at all.
P = model.initRoutingMatrix()
for r in range(6):
    P.set(jobclass[r], jobclass[r], q1, q2, 1.0)
    P.set(jobclass[r], jobclass[r], q2, q1, 1.0)
model.link(P)

    ... (13 source lines omitted; full implementation: python/examples/advanced/)

          % (Tc['Station'][i], Tc['JobClass'][i], qc[i], qn[i]))

err = float(np.max(np.abs(qn - qc)))
print('\nNC-samp vs CTMC:  max|dQ| = %.3e (importance-sampling noise)' % err)
assert err <= 5e-2, 'NC sampling does not match the exact CTMC within IS tolerance'
print("PASS: SolverNC 'sampling' matches the exact CTMC within importance-sampling noise.")
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.13.7 pas_saturation

The saturated regime, where the swap rule sets the throughput.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.

Solvers	CTMC, LDES

Table 3.62: Features of `pas_saturation`.

The example is built and solved as follows.

```
Pass-and-swap (PAS) saturation handling for large / unbounded buffers.
An order-independent service rate mu(c) usually SATURATES: beyond some per-class
count threshold tau_r, adding more class-r jobs no longer changes mu (e.g. a
compatibility / rank function saturates once each present class has one job; an
M/M/K rate saturates at K jobs). LDES exploits this: it serialises mu as a
rate table over the saturation box prod_r {0..tau_r} plus the cutoffs tau, and
the simulator clamps each class count to tau before the table lookup. The table
therefore stays COMPACT regardless of the buffer size, so a large buffer can
approximate an unbounded queue cheaply.
This example uses the Comte (thesis Sect. 4.1) compatibility queue: I=2 classes,
S=3 servers, S_1={1,2}, S_2={2,3}, rank function
    mu(A) = sum_{ s : exists present class in A compatible with s } cap_s,
so mu({1})=mu({2})=cap_1+cap_2 = 3, mu({1,2})=cap_1+cap_2+cap_3 = 5, and mu is
constant once both classes are present (cutoffs tau=[1,1]).
It checks that (i) the serialised table stays at 3 entries whatever the buffer,
(ii) LDES matches CTMC, and (iii) an unset/infinite buffer raises a
clean, actionable error (saturating rate -> suggest setCap; unbounded additive
rate -> setCap required).
# compatibility comp[s, i] = 1 iff server s serves class i; capacities cap_s
comp = np.array([[1, 0],      # server 1 -> class 1
                 [1, 1],      # server 2 -> classes 1, 2 (shared)
                 [0, 1]])     # server 3 -> class 2
cap_s = np.array([2.0, 1.0, 2.0])
lam = [1.5, 1.0]
I = comp.shape[1]
def mu_rank(c):

    ... (57 source lines omitted; full implementation: python/examples/advanced/)

        raise AssertionError('expected a finite-buffer error for ' + tag)
    except Exception as e:
        msg = [ln for ln in str(e).split('\n') if 'PAS station' in ln]
        print('  %-20s -> %s' % (tag, (msg[0] if msg else str(e))[:88]))
print('\nPASS: PAS saturation handling keeps the table compact, matches CTMC, '
      'and errors cleanly on infinite buffers.')
```

Running this edition prints

```
saturation keeps the order-independent rate table compact:
  cap=8     table entries = 3  cutoffs = [1, 1]
  cap=100   table entries = 3  cutoffs = [1, 1]
  cap=1000  table entries = 3  cutoffs = [1, 1]
CTMC solver using state space cutoff = 8 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverLDES command: common/ldes solve <tmp> -o <tmp> -s 300000 --seed 23000

open compatibility PAS, buffer=8:
  CTMC  QLen = [1.05517 0.57222]
  LDES  QLen = [1.05848 0.57269]
  LDES vs CTMC max rel|dQ| = 0.31%

infinite-buffer PAS raises a clean, actionable error:
  saturating rank      -> PAS station 'PASQueue' has no finite buffer: LDES state-space generation
    requires one. I
  unbounded additive   -> PAS station 'PASQueue' has an unbounded order-independent service rate (mu does
    not satu

PASS: PAS saturation handling keeps the table compact, matches CTMC, and errors cleanly on infinite
    buffers.
```

The rows repeat for each of the 2 solvers the script runs (CTMC, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

3.13.8 pas_selfloop

A self-looping class under pass-and-swap.

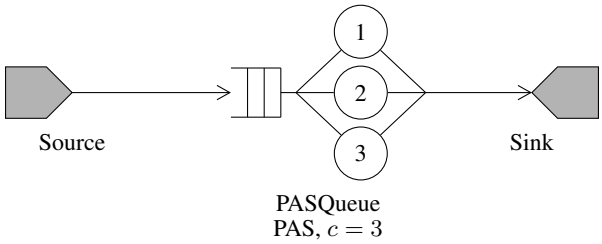

Figure 3.59: Topology of pas_selfloop: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	3: Class1 (open), Class2 (open), Class3 (open)
Scheduling	PASQueue: PAS ($c = 3$)
Arrivals	Source – Class1: Exp(0.6); Class2: Exp(0.4); Class3: Exp(0.3)
Service	PASQueue – Class1: Exp(1); Class2: Exp(1); Class3: Exp(1)
Features	finite buffer 3 at PASQueue
Solvers	CTMC

Table 3.63: Features of pas_selfloop.

The example is built and solved as follows.

```
Pass-and-swap (PAS) queue with a swapping graph that includes a self-loop.
Per Dorsman & Gardner (2024), Sect. 2.3, self-loops in the swapping graph are
permissible: a completing class-i job may take the place of another class-i job
further back in the queue. The order-independent product form (Theorem 2) still
holds and remains invariant to the swapping graph.
# Three unit-rate servers, one per class: mu(c) = number of distinct classes in c.
comp = np.eye(3)
def mu_fun(c):
    c = np.asarray(c, dtype=int)
    return float(np.sum(np.any(comp[:, c], axis=1))) if len(c) else 0.0
model = Network('PASselfloop')
source = Source(model, 'Source')
queue = Queue(model, 'PASQueue', SchedStrategy.PAS)
sink = Sink(model, 'Sink')
lam = [0.6, 0.4, 0.3]
classes = [OpenClass(model, f'Class{r+1}') for r in range(3)]
for r in range(3):
    source.setArrival(classes[r], Exp(lam[r]))
queue.setService(mu_fun)
# Swapping graph with a self-loop on class 0 and an edge (1,2) (0-based)
G = np.zeros((3, 3))
G[0, 0] = 1            # self-loop: class 0 swaps with class 0
G[1, 2] = 1
G[2, 1] = 1
queue.setSwapGraph(G)
queue.setNumberOfServers(3)
queue.setCap(3)
P = model.initRoutingMatrix()
for r in range(3):
    P[classes[r]] = Network.serialRouting(source, queue, sink)
model.link(P)
AvgTable = CTMC(model, cutoff=3).getAvgTable()
print(AvgTable)
```

Running this edition prints

```
CTMC solver using state space cutoff = 3 for open/mixed model.
```

```
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station   JobClass     QLen      Util    RespT  ResidT     ArvR     Tput
Source    Class1          0         0        0       0        0  0.47972
Source    Class2          0         0        0       0        0  0.31981
Source    Class3          0         0        0       0        0  0.23986
PASQueue  Class1    0.72167   0.15991   1.5043  1.5043  0.47972  0.47972
PASQueue  Class2    0.41993    0.1066    1.313   1.313  0.31981  0.31981
PASQueue  Class3    0.29409  0.079954   1.2261  1.2261  0.23986  0.23986
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.14 Reward models

A reward model attaches a rate to each state, or a value to each transition, and reports the expected accumulated reward. Cost per waiting job, revenue per completion and penalty per loss are the usual three, and together they turn a performance model into an economic one. Rewards may be declared over aggregate states when the detailed state space is too large to enumerate.

3.14.1 `rewardModel_aggregation`

Rewards over aggregated states, when the detailed state is too large to enumerate.

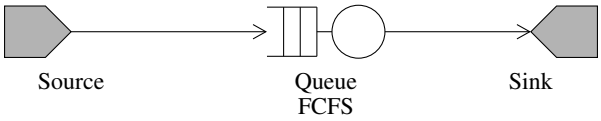

Figure 3.60: Topology of `rewardModel_aggregation`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: HighPriority (open), LowPriority (open)
Scheduling	Queue: FCFS
Arrivals	Source – HighPriority: Exp(1); LowPriority: Exp(0.8)
Service	Queue – HighPriority: Exp(3); LowPriority: Exp(3)
Features	finite buffer 4 at Queue

Table 3.64: Features of `rewardModel_aggregation`.

The example is built and solved as follows.

```
Reward Model - Aggregation Example
This example demonstrates the aggregation capabilities of the RewardState API.
The RewardStateView class supports aggregation operations:
  - total()  : Sum all values
  - max()    : Maximum value
  - min()    : Minimum value
  - count()  : Count non-zero entries
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
print('=== Reward Model - Aggregation Example ===\n')
# Model Definition
model = Network('RewardAggregationExample')
# Nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
```

```
sink = Sink(model, 'Sink')
# Queue parameters
queue.setNumberOfServers(1)
queue.setCapacity(4)  # Limit state space
# Job classes
class1 = OpenClass(model, 'HighPriority')
class2 = OpenClass(model, 'LowPriority')
# Arrivals
source.setArrival(class1, Exp(1.0))     # High priority arrival rate = 1.0
source.setArrival(class2, Exp(0.8))     # Low priority arrival rate = 0.8
# Service times

    ... (36 source lines omitted; full implementation: python/examples/advanced/)

print(f'  {"MaxClass":20s}: {R[1]:10.6f}')
print(f'  {"ClassCount":20s}: {R[2]:10.6f}')
print(f'  {"HP_Jobs":20s}: {R[3]:10.6f}')
print(f'  {"LP_Jobs":20s}: {R[4]:10.6f}')
print(f'  {"WeightedLoad":20s}: {R[5]:10.6f}')
print('\nExample completed successfully.')
```

Running this edition prints

```
=== Reward Model - Aggregation Example ===

Defining reward metrics using aggregation operations:

  TotalJobs = state.at(queue).total()
  MaxClass = state.at(queue).max()
  ClassCount = state.at(queue).count()
  HP_Jobs = state.at(queue, class1)
  LP_Jobs = state.at(queue, class2)
  WeightedLoad = 2.0 * HP + 1.0 * LP

Solving with CTMC solver...

CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
=== Steady-State Expected Rewards (Aggregation) ===
  TotalJobs            :   1.078418
  MaxClass             :   0.862690
  ClassCount           :   0.761440
  HP_Jobs              :   0.599121
  LP_Jobs              :   0.479297
  WeightedLoad         :   1.677539

Example completed successfully.
```

3.14.2 `rewardModel_mm1k`

State rewards on an M/M/1/K queue: cost per waiting job, revenue per completion, penalty per loss.

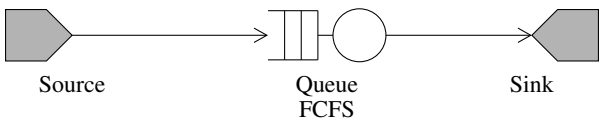

Figure 3.61: Topology of `rewardModel_mm1k`: an open network over 1 queueing station.

The example is built and solved as follows.

```
Reward Model - Basic Example
This example demonstrates the setReward feature for defining custom
reward functions on a queueing network model and computing steady-state
expected rewards using the CTMC solver.
Reward functions allow modeling various performance metrics such as:
- Queue lengths
```

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(2)
Service	Queue – Class1: Exp(3)
Features	finite buffer 3 at Queue
Solvers	CTMC

Table 3.65: Features of `rewardModel_mm1k`.

```
- Server utilization
- Blocking probabilities
- Custom cost functions
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Model Definition
# Create a simple M/M/1/K queue with finite buffer
model = Network('RewardExample')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Set finite buffer capacity
queue.setNumberOfServers(1)
queue.setCapacity(3)  # Maximum jobs in the system
# Block 2: job classes
oclass = OpenClass(model, 'Class1')
source.setArrival(oclass, Exp(2))  # Arrival rate = 2
queue.setService(oclass, Exp(3))   # Service rate = 3 (utilization ~ 0.67)
# Block 3: topology

    ... (55 source lines omitted; full implementation: python/examples/advanced/)

print(f'{"BlockingProb":>15s}: LINE = {R[2]:.6f}, Analytical = {B_analytical:.6f}, Error = {
    abs(R[2] - B_analytical):.2e}')
print(f'{"QueueCost":>15s}: LINE = {R[3]:.6f}, Analytical = {Cost_analytical:.6f}, Error = {
    abs(R[3] - Cost_analytical):.2e}')
print('\nNote: Reward functions provide a powerful way to define custom performance')
print('      metrics beyond standard measures like utilization and queue length.')
print('      The CTMC solver can compute steady-state expected values and transient')
print('      behavior of these reward functions.')
```

Running this edition prints

```
=== Reward Model - Basic Example ===

Solving with CTMC solver...

CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

=== Steady-State Expected Rewards ===
    QueueLength: 1.015385
    Utilization: 0.584615
   BlockingProb: 0.123077
      QueueCost: 2.123077
CTMC solver using state space cutoff = 10 for open/mixed model.

... (12 captured-output lines omitted; rerun the source example for the full output)

    Utilization: LINE = 0.584615, Analytical = 0.662768, Error = 7.82e-02
   BlockingProb: LINE = 0.123077, Analytical = 0.005848, Error = 1.17e-01
      QueueCost: LINE = 2.123077, Analytical = 8.070120, Error = 5.95e+00

Note: Reward functions provide a powerful way to define custom performance
      metrics beyond standard measures like utilization and queue length.
```

```
    The CTMC solver can compute steady-state expected values and transient
    behavior of these reward functions.
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

3.14.3 `rewardModel_multiclass`

Per-class rewards on a multiclass PS station.

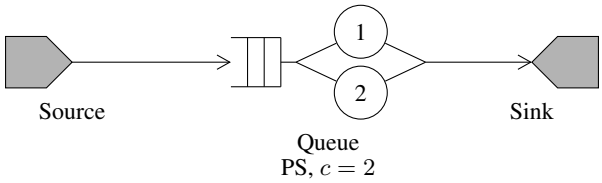

Figure 3.62: Topology of `rewardModel_multiclass`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Interactive (open), Batch (open)
Scheduling	Queue: PS ($c = 2$)
Arrivals	Source – Interactive: Exp(2); Batch: Exp(1.5)
Service	Queue – Interactive: Exp(0.5); Batch: Exp(1)
Features	finite buffer 6 at Queue

Table 3.66: Features of `rewardModel_multiclass`.

The example is built and solved as follows.

```
Reward Model - Multi-Class Example
This example demonstrates reward-based CTMC analysis in a multi-class queueing
network, showing how to define per-class metrics and analyze class-specific behavior.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
print('=== Reward Model - Multi-Class Example ===\n')
# Model Definition
model = Network('MultiClassRewardExample')
# Nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.PS)  # Processor Sharing
sink = Sink(model, 'Sink')
# Server capacity and buffer limit
queue.setNumberOfServers(2)
queue.setCapacity(6)  # Limit state space
# Job classes with different characteristics
class_interactive = OpenClass(model, 'Interactive')
class_batch = OpenClass(model, 'Batch')
# Different arrival rates
source.setArrival(class_interactive, Exp(2.0))  # Interactive: lambda = 2.0
source.setArrival(class_batch, Exp(1.5))        # Batch: lambda = 1.5
# Different service time requirements
queue.setService(class_interactive, Exp(0.5))   # Interactive: mu = 2.0 (fast)
queue.setService(class_batch, Exp(1.0))         # Batch: mu = 1.0 (slow)
# Topology (same for both classes)
P = model.initRoutingMatrix()

    ... (85 source lines omitted; full implementation: python/examples/advanced/)

est_resp_time_int = R[0] / lambda_int
est_resp_time_batch = R[2] / lambda_batch
print("\nEstimated Response Times (Little's Law):")
print(f'  Interactive: {est_resp_time_int:.6f} time units')
```

```
print(f'  Batch: {est_resp_time_batch:.6f} time units')
print('\nExample completed successfully.')
```

Running this edition prints

```
=== Reward Model - Multi-Class Example ===

Defining per-class reward metrics:

  Interactive_QLen = state.at(queue, Interactive)
  Interactive_Util = Reward.utilization(queue, Interactive)
  Batch_QLen = state.at(queue, Batch)
  Batch_Util = Reward.utilization(queue, Batch)
  Total_QLen = state.at(queue).total()
  Total_Util = Reward.utilization(queue)
  Interactive_Ratio = Interactive / Batch
  Weighted_Cost = 3.0*Interactive + 1.0*Batch
  Fairness = (Interactive > 0) AND (Batch > 0)

Solving with CTMC solver...


... (30 captured-output lines omitted; rerun the source example for the full output)

  Interactive jobs: 72.73% of queue
  Batch jobs: 27.27% of queue

Estimated Response Times (Little's Law):
  Interactive: 1.977624 time units
  Batch: 0.988812 time units

Example completed successfully.
```

3.14.4 `rewardModel_templates`

The reward templates that cover the common cases without writing a reward function.

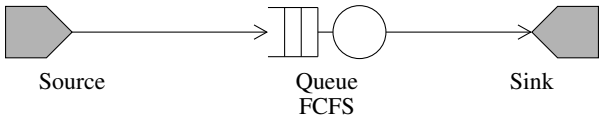

Figure 3.63: Topology of `rewardModel_templates`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1.5)
Service	Queue – Class1: Exp(2)
Features	finite buffer 10 at Queue

Table 3.67: Features of `rewardModel_templates`.

The example is built and solved as follows.

```
Reward Model - Templates Example
This example demonstrates the use of Reward template factory methods
for quickly defining common metrics like queue length, utilization, and blocking.
The Reward templates provide a convenient way to define metrics
without writing lambda expressions.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
print('=== Reward Model - Templates Example ===\n')
# Model Definition
model = Network('RewardTemplatesExample')
# Nodes
```

```
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Queue parameters
queue.setNumberOfServers(1)
queue.setCapacity(10)
# Job class
oclass = OpenClass(model, 'Class1')
source.setArrival(oclass, Exp(1.5))  # Arrival rate = 1.5
queue.setService(oclass, Exp(2))     # Service rate = 2 (rho = 0.75)
# Topology
model.link(Network.serial_routing([source, queue, sink]))
# Define Rewards Using Templates
print('Defining reward metrics using Reward templates:\n')
# Template 1: Queue Length

    ... (39 source lines omitted; full implementation: python/examples/advanced/)

B_analytical = pi[K]                                    # Blocking probability
print(f'{"QueueLength":25s}: LINE = {R[0]:10.6f}, Analytical = {L_analytical:10.6f}, Error =
    {abs(R[0] - L_analytical):.2e}')
print(f'{"Utilization":25s}: LINE = {R[2]:10.6f}, Analytical = {U_analytical:10.6f}, Error =
    {abs(R[2] - U_analytical):.2e}')
print(f'{"BlockingProb":25s}: LINE = {R[4]:10.6f}, Analytical = {B_analytical:10.6f}, Error =
     {abs(R[4] - B_analytical):.2e}')
print('\nExample completed successfully.')
print('  All template-based rewards match analytical results!')
```

Running this edition prints

```
=== Reward Model - Templates Example ===

Defining reward metrics using Reward templates:

  QueueLength = Reward.queue_length(queue)
  QueueLength_Class1 = Reward.queue_length(queue, oclass)
  Utilization = Reward.utilization(queue)
  Utilization_Class1 = Reward.utilization(queue, oclass)
  BlockingProb = Reward.blocking(queue)

Solving with CTMC solver...

CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
    inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
=== Steady-State Expected Rewards (Templates) ===

... (5 captured-output lines omitted; rerun the source example for the full output)


=== Comparison with M/M/1 Analytical Results ===
QueueLength              : LINE =   2.514926, Analytical =   2.514926, Error = 2.22e-15
Utilization              : LINE =   0.738976, Analytical =   0.738976, Error = 1.11e-16
BlockingProb             : LINE =   0.014699, Analytical =   0.014699, Error = 6.59e-17

Example completed successfully.
  All template-based rewards match analytical results!
```

3.15 Agent models

The agent interface builds a network from the point of view of the entities that move through it rather than from the stations. The resulting models are ordinary networks – Jackson networks, closed multiclass networks, G-networks with their negative customers – and are solved by the same solvers.

3.15.1 `ag_closed_network`

The closed counterpart.

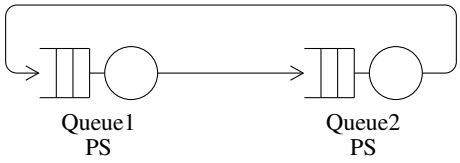

Figure 3.64: Topology of `ag_closed_network`: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	1: Class1 (closed, $N = 10$)
Scheduling	Queue1: PS; Queue2: PS
Service	Queue1 – Class1: Exp(2) Queue2 – Class1: Exp(1)
Solvers	MVA, CTMC

Table 3.68: Features of `ag_closed_network`.

The example is built and solved as follows.

```
Closed Network with AG
This example demonstrates SolverAG, the agent-based solver, on a closed queueing network
with processor-sharing (PS) queues using the INAP algorithm.
The solver decomposes the network
into interacting stochastic processes and uses fixed-point iteration
to compute equilibrium measures.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Parameters
N = 10       # Number of jobs
mu1 = 2.0    # Service rate at queue 1
mu2 = 1.0    # Service rate at queue 2
# Create model: Queue1 <-> Queue2 (closed loop)
model = Network('Closed-2Q')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
cclass = ClosedClass(model, 'Class1', N, queue1)
queue1.setService(cclass, Exp(mu1))
queue2.setService(cclass, Exp(mu2))
model.link(Network.serial_routing([queue1, queue2]))
# Solve with AG using INAP method
print('=== Closed Network ===\n')
solver_inap = AG(model, 'inap')
avg_table_inap = solver_inap.get_avg_table()
print('AG (method=inap):')
print(avg_table_inap)

    ... (5 source lines omitted; full implementation: python/examples/advanced/)

# Solve with CTMC for exact results (if state space is small)
if N <= 20:
    solver_ctmc = CTMC(model)
    avg_table_ctmc = solver_ctmc.get_avg_table()
    print('\nCTMC:')
    print(avg_table_ctmc)
```

Running this edition prints

```
=== Closed Network ===

python/line_solver/solvers/solver_ag/algorithms/ag_inap.py:1569: UserWarning: INAP is designed for open
     networks. Closed network results are approximate; consider using MVA instead.
  warnings.warn(
AG (method=inap):
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Queue1   Class1    2.4134  0.72839  1.6567  1.6567  0.99485   1.4568
Queue2   Class1    8.1985  0.99485  8.2409  8.2409   1.4568  0.99485
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
```

```
MVA:
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   Class1    0.99463  0.49976  0.99511  0.99511  0.99951  0.99951
Queue2   Class1     9.0054  0.99951   9.0098   9.0098  0.99951  0.99951
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.

CTMC:
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   Class1    0.99463  0.49976  0.99511  0.99511  0.99951  0.99951
Queue2   Class1     9.0054  0.99951   9.0098   9.0098  0.99951  0.99951
```

The rows repeat for each of the 2 solvers the script runs (MVA, CTMC), which is the point of the example: the same model read by methods of different cost and different exactness.

3.15.2 ag_gnetwork

A G-network, with the negative customers that remove jobs on arrival.

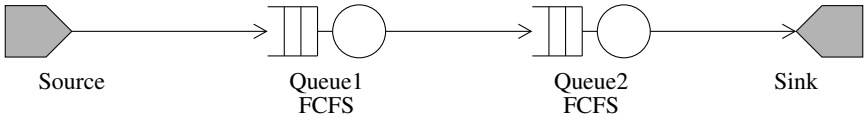

Figure 3.65: Topology of ag_gnetwork: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	2: Positive (open), Negative (signal)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Positive: Exp(1); Negative: Exp(0.3)
Service	Queue1 – Positive: Exp(2); Negative: Exp(2) Queue2 – Positive: Exp(3); Negative: Exp(3)

Table 3.69: Features of ag_gnetwork.

The example is built and solved as follows.

```
G-Network (Gelenbe Network) with Negative Customers
This example demonstrates SolverAG, the agent-based solver, on a G-network with negative
    customers.
Negative customers (signals) remove jobs from queues when they arrive,
modeling job cancellations or service interrupts.
Network topology:
  - Source generates positive customers (Class1) and negative signals (Class2)
  - Positive customers flow: Source -> Queue1 -> Queue2 -> Sink
  - Negative signals target Queue2, removing jobs from it
Reference: Gelenbe, E. (1991). "Product-form queueing networks with
           negative and positive customers", Journal of Applied Probability
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Parameters
lambda_pos = 1.0   # Positive customer arrival rate
lambda_neg = 0.3   # Negative signal arrival rate
mu1 = 2.0          # Service rate at Queue1
mu2 = 3.0          # Service rate at Queue2
# Create model
model = Network('GNetwork-Example')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Positive customer class (normal jobs)
posClass = OpenClass(model, 'Positive')
source.setArrival(posClass, Exp(lambda_pos))
```

```
    ... (18 source lines omitted; full implementation: python/examples/advanced/)

# Solve with AG using INAP method
print('=== G-Network with Negative Customers ===\n')
solver_ag = AG(model, 'inap')
avg_table = solver_ag.get_avg_table()
print('AG (method=inap):')
print(avg_table)
```

Running this edition prints

```
=== G-Network with Negative Customers ===

AG (method=inap):
Station  JobClass      QLen     Util     RespT    ResidT     ArvR     Tput
Source   Positive        0        0         0         0        0        1
Source   Negative        0        0         0         0        0      0.3
Queue1   Positive  0.76923  0.43478   0.88462   0.88462        1  0.86957
Queue1   Negative        0        0         0         0      0.3        0
Queue2   Positive  0.40816  0.28986   0.46939   0.46939  0.86957  0.86957
```

3.15.3 ag_jackson_network

A Jackson network built agent by agent.

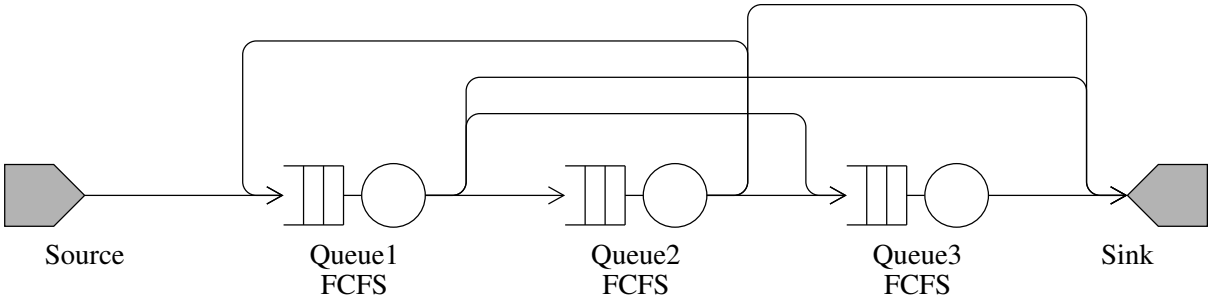

Figure 3.66: Topology of ag_jackson_network: an open network over 3 queueing stations.

Nodes	5: Source, Queue × 3, Sink
Classes	1: Class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Queue1 – Class1: Exp(2) Queue2 – Class1: Exp(3) Queue3 – Class1: Exp(2.5)
Solvers	MVA

Table 3.70: Features of ag_jackson_network.

The example is built and solved as follows.

```
Jackson Network with AG
This example demonstrates SolverAG, the agent-based solver, on an open Jackson network with
probabilistic routing. The solver models job transfers between
queues as synchronization actions between interacting processes.
Reference: Jackson, J.R. (1957). "Networks of waiting lines"
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Parameters
lam = 1.0   # External arrival rate
mu = [2.0, 3.0, 2.5]  # Service rates
# Routing probabilities
p12 = 0.4  # Queue1 -> Queue2
p13 = 0.3  # Queue1 -> Queue3
```

```
p1s = 0.3  # Queue1 -> Sink
p21 = 0.2  # Queue2 -> Queue1
p23 = 0.3  # Queue2 -> Queue3
p2s = 0.5  # Queue2 -> Sink
p3s = 1.0  # Queue3 -> Sink
# Create model
model = Network('Jackson-3Q')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Class1')

    ... (20 source lines omitted; full implementation: python/examples/advanced/)

print(avg_table_inap)
# Solve with MVA for comparison
solver_mva = MVA(model)
avg_table_mva = solver_mva.get_avg_table()
print('\nMVA:')
print(avg_table_mva)
```

Running this edition prints

```
=== Jackson Network ===

AG (method=inap):
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source   Class1          0        0        0        0        0        1
Queue1   Class1     1.1905  0.54349   1.0953   1.1905    1.087    1.087
Queue2   Class1    0.16949  0.14492  0.38983  0.16949  0.43479  0.43477
Queue3   Class1    0.22342  0.18262  0.48937  0.22341  0.45653  0.45655
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

MVA:
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source   Class1          0        0        0        0        0        1
Queue1   Class1     1.1905  0.54348   1.0952   1.1905    1.087    1.087
Queue2   Class1    0.16949  0.14493  0.38983  0.16949  0.43478  0.43478
Queue3   Class1     0.2234  0.18261  0.48936   0.2234  0.45652  0.45652
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.15.4 `ag_multiclass_closed`

The multiclass closed version.

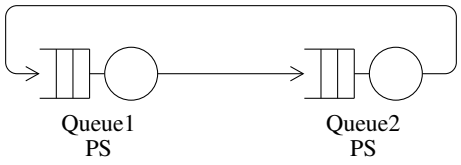

Figure 3.67: Topology of `ag_multiclass_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	2: Class1 (closed, $N = 5$), Class2 (closed, $N = 3$)
Scheduling	Queue1: PS; Queue2: PS
Service	Queue1 – Class1: Exp(2); Class2: Exp(1.5) Queue2 – Class1: Exp(1); Class2: Exp(0.8)
Solvers	MVA

Table 3.71: Features of `ag_multiclass_closed`.

The example is built and solved as follows.

```
Multiclass Closed Network with AG
This example demonstrates SolverAG, the agent-based solver, on a multiclass closed queueing
network. The solver handles multiple job classes by creating
separate processes for each (station, class) pair.
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Parameters
N1 = 5       # Number of class 1 jobs
N2 = 3       # Number of class 2 jobs
mu1_q1 = 2.0  # Service rate for class 1 at queue 1
mu2_q1 = 1.5  # Service rate for class 2 at queue 1
mu1_q2 = 1.0  # Service rate for class 1 at queue 2
mu2_q2 = 0.8  # Service rate for class 2 at queue 2
# Create model
model = Network('Multiclass-Closed')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
class1 = ClosedClass(model, 'Class1', N1, queue1)
class2 = ClosedClass(model, 'Class2', N2, queue1)
queue1.setService(class1, Exp(mu1_q1))
queue1.setService(class2, Exp(mu2_q1))
queue2.setService(class1, Exp(mu1_q2))
queue2.setService(class2, Exp(mu2_q2))
# Simple routing: each class cycles through both queues
P = model.initRoutingMatrix()
P.set(class1, class1, queue1, queue2, 1.0)

    ... (9 source lines omitted; full implementation: python/examples/advanced/)

print(avg_table_inap)
# Solve with MVA for comparison
solver_mva = MVA(model)
avg_table_mva = solver_mva.get_avg_table()
print('\nMVA:')
print(avg_table_mva)
```

Running this edition prints

```
=== Multiclass Closed Network ===

python/line_solver/solvers/solver_ag/algorithms/ag_inap.py:1569: UserWarning: INAP is designed for open
     networks. Closed network results are approximate; consider using MVA instead.
  warnings.warn(
AG (method=inap):
Station  JobClass    QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Class1    1.6243  0.67984  1.1946  1.1946  0.95678   1.3597
Queue1   Class2    1.1225  0.62483  1.1977  1.1977  0.70154  0.93725
Queue2   Class1    3.6418  0.95678  3.8062  3.8062   1.3597  0.95678
Queue2   Class2    1.9846  0.87692  2.8289  2.8289  0.93725  0.70154
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.

MVA:
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Class1    0.63167  0.31287  1.0095  1.0095  0.62574  0.62574
Queue1   Class2    0.39711  0.19837  1.3346  1.3346  0.29755  0.29755
Queue2   Class1     4.3683  0.62574   6.981   6.981  0.62574  0.62574
Queue2   Class2     2.6029  0.37194  8.7476  8.7476  0.29755  0.29755
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.15.5 `ag_tandem_open`

An open tandem written through the agent interface.
The example is built and solved as follows.

```
Open Tandem Queue with AG (INAP method)
```

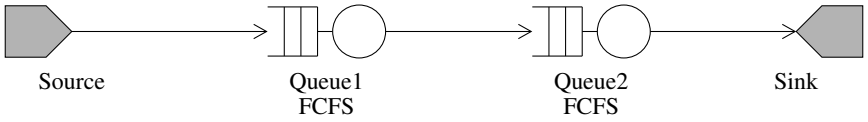

Figure 3.68: Topology of `ag_tandem_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: Class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.5)
Service	Queue1 – Class1: Exp(1) Queue2 – Class1: Exp(1.5)
Solvers	MVA

Table 3.72: Features of `ag_tandem_open`.

```
This example demonstrates SolverAG with the INAP method on an open tandem queueing network
(M/M/1 -> M/M/1) using the agent-based INAP method.
Reference: Marin and Rota-Bulo', "A Mean-Field Analysis of a Class of
           Interactive Distributed Systems", MASCOTS 2009
Copyright (c) 2012-2025, Imperial College London
All rights reserved.
# Parameters
arrival_rate = 0.5   # Arrival rate
service_rate1 = 1.0  # Service rate at queue 1
service_rate2 = 1.5  # Service rate at queue 2
# Create model: Source -> Queue1 -> Queue2 -> Sink
model = Network('Tandem-MM1')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Class1')
source.setArrival(oclass, Exp(arrival_rate))
queue1.setService(oclass, Exp(service_rate1))
queue2.setService(oclass, Exp(service_rate2))
model.link(Network.serial_routing([source, queue1, queue2, sink]))
# Analytical M/M/1 solution
U1_exact = arrival_rate / service_rate1
U2_exact = arrival_rate / service_rate2
Q1_exact = U1_exact / (1 - U1_exact)

    ... (13 source lines omitted; full implementation: python/examples/advanced/)

# It falls back to INAP with a warning in the JAR version
# Solve with MVA for comparison
solver_mva = MVA(model)
avg_table_mva = solver_mva.get_avg_table()
print('\nMVA:')
print(avg_table_mva)
```

Running this edition prints

```
=== Open Tandem Queue (M/M/1 -> M/M/1) ===

Analytical (M/M/1):
  Queue1: U=0.5000, Q=1.0000, R=2.0000
  Queue2: U=0.3333, Q=0.5000, R=1.0000

AG (method=inap):
Station  JobClass  QLen     Util  RespT  ResidT  ArvR  Tput
Source   Class1       0        0      0       0     0   0.5
Queue1   Class1       1      0.5      2       2   0.5   0.5
Queue2   Class1     0.5  0.33333      1       1   0.5   0.5
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.

MVA:
```

```
Station  JobClass  QLen     Util  RespT  ResidT  ArvR  Tput
Source   Class1       0        0      0       0     0   0.5
Queue1   Class1       1      0.5      2       2   0.5   0.5
Queue2   Class1     0.5  0.33333      1       1   0.5   0.5
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

3.15.6 `ag_tandem_phasetype`

An open tandem whose two queues share a mean service time and differ only in variability (Erlang-2 against a hyperexponential), solved by the RCAT methods, which carry the phase-type law exactly where an M/M/1 reading sees no difference.

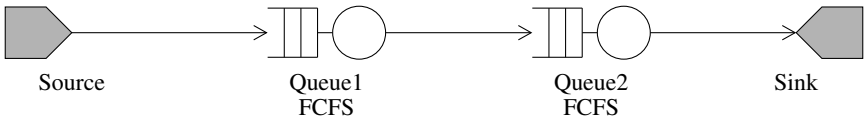

Figure 3.69: Topology of `ag_tandem_phasetype`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: Class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.5)
Service	Queue1 – Class1: Erlang(2,2) Queue2 – Class1: HyperExp(0.99,0.01;1.14,0.07584)

Table 3.73: Features of `ag_tandem_phasetype`.

The example is built and solved as follows.

```
Open Tandem Queue with Phase-Type Service, solved by the agent-based solver
This example demonstrates that the agent-based methods represent a phase-type service
law exactly rather than collapsing it to its mean rate. Each component of the
agent decomposition is a QBD whose level is the queue length and whose phase is
the pair (arrival phase, service phase), so the first station -- an isolated
M/PH/1, since it sees the Poisson source directly -- comes out at the
Pollaczek-Khinchine mean whatever the reversed-rate iteration does. Both
stations here have the same mean service time and differ only in their
variability, which is exactly what an M/M/1 reading cannot see.
References: Casale and Harrison, "AutoCAT: Automated Product-Form Solution of
            Stochastic Models", Stochastic Models 27, 2013
            Neuts, "Matrix-Geometric Solutions in Stochastic Models", 1981
Copyright (c) 2012-2026, Imperial College London
All rights reserved.
# Parameters
arrival_rate = 0.5    # Arrival rate
mean_service = 1.0    # Mean service time at both queues
scv1 = 0.5            # Erlang-2 service at queue 1 (less variable than exponential)
scv2 = 4.0            # HyperExp service at queue 2 (more variable)
# Create model: Source -> Queue1 -> Queue2 -> Sink
model = Network('Tandem-MPH1')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Class1')

    ... (12 source lines omitted; full implementation: python/examples/advanced/)

print(AG(model, 'inap').get_avg_table())
# 'inapinf' additionally drops the maxStates truncation, solving each open
# component on its infinite state space through Neuts' rate matrix R. That
```

```
# matters most at Queue2, whose service law has the heavier tail.
print('\nAG (method=inapinf):')
print(AG(model, 'inapinf').get_avg_table())
```

Running this edition prints

```
=== Open Tandem with Phase-Type Service ===

Queue1 is an isolated M/Er2/1, so its exact mean queue length is the
Pollaczek-Khinchine value 0.875000. The M/M/1 reading would be 1.000000.

AG (method=inap):
Station  JobClass   QLen  Util   RespT  ResidT  ArvR  Tput
Source   Class1        0     0       0       0     0   0.5
Queue1   Class1    0.875   0.5    1.75    1.75   0.5   0.5
Queue2   Class1     1.75   0.5  3.4999  3.4999   0.5   0.5

AG (method=inapinf):
Station  JobClass   QLen  Util  RespT  ResidT  ArvR  Tput
Source   Class1        0     0      0       0     0   0.5
Queue1   Class1    0.875   0.5   1.75    1.75   0.5   0.5
Queue2   Class1     1.75   0.5    3.5     3.5   0.5   0.5
```

3.16 Large-scale models

When the state space cannot be enumerated, the exact solvers are out and the question becomes how much accuracy the approximations retain. The script here is a large closed PS network solved by several approximate methods.

3.16.1 `largescale_tbi`

A large closed PS network, and the accuracy the approximate solvers retain on it.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
                    Network, Queue, SchedStrategy, VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.SILENT)

model = Network('tbi_largescale')
rng = np.random.default_rng(1)

Q = [Queue(model, 'Q%d' % (i + 1), SchedStrategy.PS) for i in range(M)]
D = {}
for i in range(M):
    for j in range(M):
        if i != j:
            D[(i, j)] = Delay(model, 'D%d_%d' % (i + 1, j + 1))

job = ClosedClass(model, 'C1', N, Q[0])
lam = np.zeros((M, M))
for i in range(M):
    Q[i].setService(job, Erlang.fitMeanAndOrder(1.0 / (1 + 3 * rng.random()), 4))
    for j in range(M):
        if i != j:
            D[(i, j)].setService(
                job, Erlang.fitMeanAndOrder(0.2 + 2 * rng.random(), KPH))
            lam[i, j] = rng.random()

P = model.initRoutingMatrix()
for i in range(M):
    pr = lam[i, :] / lam[i, :].sum()

    ... (23 source lines omitted; full implementation: python/examples/advanced/)
```

```
print(avg_table.head(M))    # queue-length summary at the queueing stations

# For comparison, the undecomposed solution of the same model:
#   solver = FLD(model, method='closing', timespan=[0, TEND], stiff=True)
# takes several minutes on the same machine (about 80 seconds already at
# M=8, and beyond 10 minutes at M=12).
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

3.17 Custom solvers

A solver is a class with an analyzer and a feature set. The feature set is a claim about what the solver can represent, and the dispatcher uses it to decide whether a model may be given to it; the analyzer computes the metrics. These two scripts show the smallest pair that the dispatcher will accept.

3.17.1 `solver_custom`

A user-defined solver registered with the dispatcher.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
SOLVER_CUSTOM  The bare solution algorithm of the custom-solver template.
[Q,U,R,T,C,X] = solver_custom(sn, options)
This is the file to fill in: it receives the NetworkStruct and the options and
must return the six average-metric matrices. The template returns zeros and says
so, exactly as the MATLAB twin does.
def solver_custom(sn, options=None):
    M = sn.nstations                                    # number of stations
    K = sn.nclasses                                     # number of classes
    N = np.asarray(sn.njobs, dtype=float).ravel()       # job populations
    rates = sn.rates                                    # arrival and service rates
    V = sn.visits                                       # visits
    QN = np.zeros((M, K))
    UN = np.zeros((M, K))
    RN = np.zeros((M, K))
    TN = np.zeros((M, K))
    CN = np.zeros((1, K))
    XN = np.zeros((1, K))
    print('The solution algorithm needs to be implemented in solver_custom.py: '
          'returning with no result.')
    return QN, UN, RN, TN, CN, XN
```

The captured run completed successfully and produced no console output.

3.17.2 `solver_custom_analyzer`

Its analyzer, the function that actually computes the metrics.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
SOLVER_CUSTOM_ANALYZER  Everything around the solution algorithm.
[Q,U,R,T,C,X,runtime] = solver_custom_analyzer(sn, options)
Any activity prior to or after launching the solution algorithm belongs here:
the template times the call and scrubs the NaNs the algorithm may leave behind.
```

```
def solver_custom_analyzer(sn, options=None):
    t0 = time.time()
    print('Any activity prior or after launching the solution algorithm needs '
          'to be implemented in solver_custom_analyzer.py.')
    QN, UN, RN, TN, CN, XN = solver_custom(sn, options)
    out = []
    for m in (QN, UN, RN, TN, CN, XN):
        m = np.asarray(m, dtype=float)
        m[np.isnan(m)] = 0.0
        out.append(m)
    return tuple(out) + (time.time() - t0,)
```

The captured run completed successfully and produced no console output.

Chapter 4

Model gallery

The gallery is a catalogue of small named models, one function each, returning a model rather than solving it. It exists twice over: as documentation, because a name in Kendall notation is the fastest way to find the model one means, and as the cross-codebase parity suite, because the same 71 functions exist in all four codebases and their metrics are compared numerically.

The models live in `python/examples/gallery/`, one file per model. A gallery entry never prints: it builds the model and returns it, so the caller chooses the solver.

```
def gallery_mm1():
    model = Network('M/M/1')
    # Block 1: nodes
    source = Source(model, 'mySource')
    queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
    sink = Sink(model, 'mySink')
    # Block 2: classes
    oclass = OpenClass(model, 'myClass')
    source.setArrival(oclass, Exp(1))
    queue.setService(oclass, Exp(2))
    # Block 3: topology
    model.link(Network.serial_routing([source, queue, sink]))
    return model
```

4.1 Reading the names

The single-station entries are named in Kendall notation, with the distributions abbreviated: `m` exponential, `d` deterministic, `erl` Erlang, `hyp` hyperexponential, `aph` acyclic phase-type, `cox` Coxian, `me` matrix-exponential, `map` Markovian arrival process, `mmap` marked MAP, `gam` Gamma, `par` Pareto, `u` uniform. So `gallery_erlm1` is an Erl/M/1 queue and `gallery_mhypk` an M/HyperExp/k. A trailing `_ps` replaces FCFS by processor sharing, `k` in place of `1` makes the station multiserver, and the suffixes `_tandem`, `_linear`, `_reentrant`, `_feedback`, `_multiclass` and `_prio` describe the topology or the workload around the station.

The remaining entries are named after what they model rather than after a queueing notation: `gallery_cqn` and `gallery_repairmen` for the closed archetypes, `gallery_fj_open` and `gallery_fj_closed` for fork-join, `gallery_fcr` for a finite capacity region, `gallery_cache_lru` and `gallery_cache_routing` for caches, `gallery_renv_breakdown` for a random environment, `gallery_lqn_basic`, `gallery_lqn_workflows` and `gallery_lqn_random` for layered models, `gallery_multitier` and `gallery_multitier_storage` for multitier applications, `gallery_lukumar_reentrant` for the Lu-Kumar re-entrant line, and `gallery_qn_random` for a generated network.

4.2 The catalogue

The complete list, in the order in which the models are registered:

```
gallery_aphm1
gallery_cache_lru
gallery_cache_routing
gallery_coxm1
gallery_cqn
gallery_cqn_multiclass
gallery_detm1
gallery_dm1
gallery_erldk
gallery_erlerl1
gallery_erlerl1_reentrant
gallery_erlm1
gallery_erlm1_ps
gallery_erlm1_reentrant
gallery_erlm1ps
gallery_fcr
gallery_fj_closed
gallery_fj_open
gallery_fj_quorum
gallery_gamm1
gallery_hyperl1_feedback
gallery_hyperl1_reentrant
gallery_hyperlk
gallery_hyphyp1_linear
gallery_hyphyp1_reentrant
gallery_hyphyp1_tandem
gallery_hypm1
gallery_hypm1_reentrant
gallery_lqn_basic
gallery_lqn_random
gallery_lqn_workflows
gallery_lukumar_reentrant
gallery_mapm1
gallery_mapmk
gallery_mdk
gallery_merl1
gallery_merl1_linear
gallery_merl1_reentrant
gallery_merl1_tandem
gallery_merlk
gallery_mhyp1
gallery_mhyp1_linear
gallery_mhyp1_reentrant
gallery_mhyp1_tandem
gallery_mhypk
gallery_mm1
gallery_mm1_feedback
gallery_mm1_linear
gallery_mm1_multiclass
gallery_mm1_prio
gallery_mm1_ps
gallery_mm1_ps_feedback
gallery_mm1_ps_multiclass
gallery_mm1_ps_reentrant
gallery_mm1_reentrant
gallery_mm1_tandem
gallery_mm1_tandem_multiclass
gallery_mm1k
gallery_mmap1
gallery_mmap1_multiclass
gallery_mmapk
gallery_mmk
gallery_mpar1
gallery_multitier
gallery_multitier_storage
gallery_parm1
gallery_qn_random
gallery_renv_breakdown
gallery_repairmen
gallery_replayerm1
gallery_um1
```

4.2.1 `gallery_aphm1`

The APH/M/1 queue: acyclic phase-type interarrival times, exponential service, and a single server.

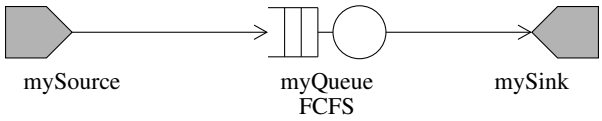

Figure 4.1: Topology of `gallery_aphm1`: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('APH/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
```

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: APH
Service	myQueue – myClass: Exp(2)

Table 4.2: Features of `gallery_aphm1`.

```
source.setArrival(oclass, APH.fit_central(1, 0.99, 1.999))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_aphm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: APH/M/1
```

4.2.2 `gallery_cache_lru`

A cache node under least-recently-used replacement, read by a closed class.

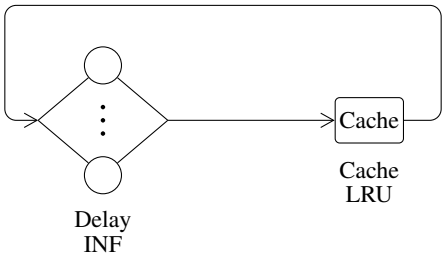

Figure 4.2: Topology of `gallery_cache_lru`: a closed network over 1 queueing station with a cache node.

Nodes	2: Delay, Cache
Classes	3: JobClass (closed, $N = 1$), HitClass (closed, $N = 0$), MissClass (closed, $N = 0$)
Scheduling	Delay: INF
Service	Delay – JobClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Features	drop rule {'HitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at Delay; cache: LRU replacement, 5 items, capacity 2; class switching on 2 links

Table 4.3: Features of `gallery_cache_lru`.

The example is built and solved as follows.

```
"""Closed cache model with LRU replacement (n=5 items, m=2 slots)."""
model = Network('Cache-LRU')
n = 5
m = 2
delay = Delay(model, 'Delay')
cacheNode = Cache(model, 'Cache', n, m, ReplacementStrategy.LRU)
jobClass = ClosedClass(model, 'JobClass', 1, delay, 0)
hitClass = ClosedClass(model, 'HitClass', 0, delay, 0)
missClass = ClosedClass(model, 'MissClass', 0, delay, 0)
delay.setService(jobClass, Exp(1))
pAccess = DiscreteSampler([1.0 / n] * n)
cacheNode.setRead(jobClass, pAccess)
```

```
cacheNode.setHitClass(jobClass, hitClass)
cacheNode.setMissClass(jobClass, missClass)
P = model.init_routing_matrix()
P.set(jobClass, jobClass, delay, cacheNode, 1.0)
P.set(hitClass, jobClass, cacheNode, delay, 1.0)
P.set(missClass, jobClass, cacheNode, delay, 1.0)
model.link(P)
return model
model = gallery_cache_lru()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.3 `gallery_cache_routing`

A cache whose hit and miss outcomes route the job to different downstream stations.

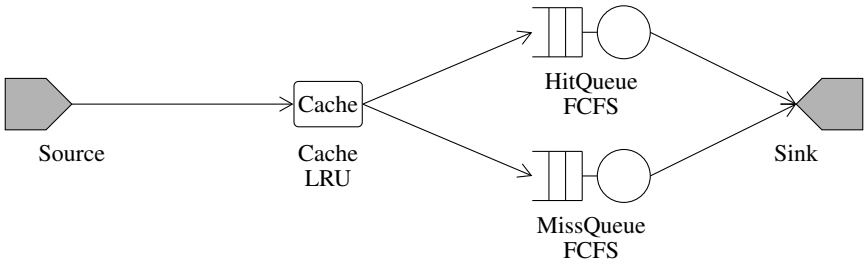

Figure 4.3: Topology of `gallery_cache_routing`: an open network over 2 queueing stations with a cache node.

Nodes	5: Source, Queue × 2, Cache, Sink
Classes	3: InitClass (open), HitClass (open), MissClass (open)
Scheduling	HitQueue: FCFS; MissQueue: FCFS
Arrivals	Source – InitClass: Exp(1); HitClass: Disabled; MissClass: Disabled
Service	HitQueue – InitClass: Disabled; HitClass: Exp(2); MissClass: Disabled MissQueue – InitClass: Disabled; HitClass: Disabled; MissClass: Exp(1)
Features	cache: LRU replacement, 4 items, capacity 2; drop rule {'InitClass': 'waitingQueue', 'MissClass': 'waitingQueue'} at HitQueue; drop rule {'InitClass': 'waitingQueue', 'HitClass': 'waitingQueue'} at MissQueue; class switching on 4 links

Table 4.4: Features of `gallery_cache_routing`.

The example is built and solved as follows.

```
"""Open cache with hit/miss routed to distinct queues."""
model = Network('Cache-Routing')
n = 4
m = 2
source = Source(model, 'Source')
cacheNode = Cache(model, 'Cache', n, m, ReplacementStrategy.LRU)
hitQueue = Queue(model, 'HitQueue', SchedStrategy.FCFS)
missQueue = Queue(model, 'MissQueue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
jobClass = OpenClass(model, 'InitClass', 0)
hitClass = OpenClass(model, 'HitClass', 0)
missClass = OpenClass(model, 'MissClass', 0)
source.setArrival(jobClass, Exp(1))
hitQueue.setService(hitClass, Exp(2.0))
missQueue.setService(missClass, Exp(1.0))
pAccess = DiscreteSampler([1.0 / n] * n)
```

```
cacheNode.setRead(jobClass, pAccess)
cacheNode.setHitClass(jobClass, hitClass)
cacheNode.setMissClass(jobClass, missClass)
P = model.init_routing_matrix()
P.set(jobClass, jobClass, source, cacheNode, 1.0)
P.set(hitClass, hitClass, cacheNode, hitQueue, 1.0)
P.set(hitClass, hitClass, hitQueue, sink, 1.0)
P.set(missClass, missClass, cacheNode, missQueue, 1.0)
P.set(missClass, missClass, missQueue, sink, 1.0)
model.link(P)
return model
model = gallery_cache_routing()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.4 `gallery_coxm1`

The Coxian/M/1 queue: Coxian interarrival times, exponential service, and a single server.

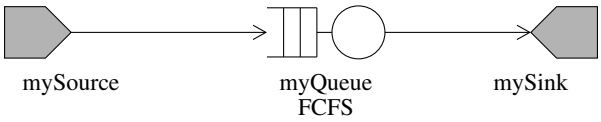

Figure 4.4: Topology of `gallery_coxm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Coxian
Service	myQueue – myClass: Exp(2)

Table 4.5: Features of `gallery_coxm1`.

The example is built and solved as follows.

```
model = Network('Cox/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Coxian.fit_central(1, 0.99, 1.999))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_coxm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Cox/M/1
```

4.2.5 gallery_cqn

The closed queueing network archetype: a fixed job population circulating between a Delay station and a queue.

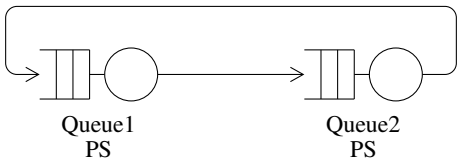

Figure 4.5: Topology of gallery_cqn: a closed network over 2 queueing stations.

Nodes	2: Queue × 2
Classes	1: Class1 (closed, $N = 3$)
Scheduling	Queue1: PS; Queue2: PS
Service	Queue1 – Class1: Exp(0.8271) Queue2 – Class1: Exp(0.4381)

Table 4.6: Features of gallery_cqn.

The example is built and solved as follows.

```
model = Network('Single-class CQN')
# Block 1: nodes
random.seed(seed)
station = []
for i in range(M):
    station.append(Queue(model, f'Queue{i+1}', SchedStrategy.PS))
if use_delay:
    station.append(Delay(model, 'Delay1'))
# Block 2: classes
jobclass = ClosedClass(model, 'Class1', round(random.random() * M + 2), station[0], 0)
for i in range(M):
    station[i].setService(jobclass, Exp.fit_mean(random.random() + i + 1))
if use_delay:
    station[-1].setService(jobclass, Exp.fit_mean(2.0))
# Block 3: topology
P = model.init_routing_matrix()
P[jobclass, jobclass] = Network.serial_routing(station)
model.link(P)
return model
model = gallery_cqn()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Single-class CQN
```

4.2.6 gallery_cqn_multiclass

The closed archetype with several classes, each with its own population.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 5$), Class2 (closed, $N = 5$)
Scheduling	Queue 1: PS; Delay 1: INF
Service	Queue 1 – Class1: Exp(0.03226); Class2: Exp(0.07143) Delay 1 – Class1: Exp(0.04762); Class2: Exp(0.08333)

Table 4.7: Features of gallery_cqn_multiclass.

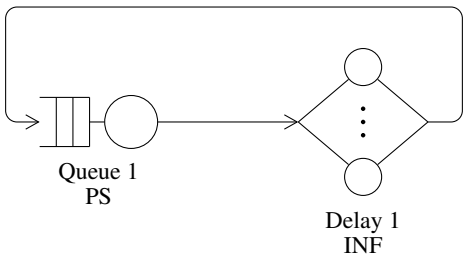

Figure 4.6: Topology of `gallery_cqn_multiclass`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
random.seed(seed)
model = Network('Multi-class CQN')
# Block 1: nodes
node = []
for i in range(m):
    node.append(Queue(model, f'Queue {i+1}', SchedStrategy.PS))
if wantdelay:
    node.append(Delay(model, 'Delay 1'))
# Block 2: classes
jobclass = []
for s in range(r):
    jobclass.append(ClosedClass(model, f'Class{s+1}', 5, node[0], 0))
for s in range(r):
    for i in range(m):
        node[i].setService(jobclass[s], Exp.fit_mean(round(50 * random.random())))
    if wantdelay:
        node[-1].setService(jobclass[s], Exp.fit_mean(round(100 * random.random())))
# Block 3: topology
P = model.init_routing_matrix()
for s in range(r):
    P[jobclass[s], jobclass[s]] = Network.serial_routing(node)
model.link(P)
return model
model = gallery_cqn_multiclass()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Multi-class CQN
```

4.2.7 `gallery_detm1`

The D/M/1 queue: deterministic interarrival times, exponential service, and a single server.

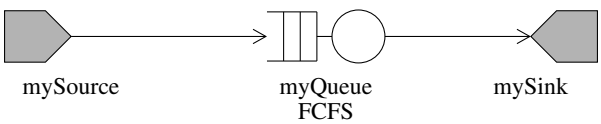

Figure 4.7: Topology of `gallery_detm1`: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('D/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
```

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Det(1)
Service	myQueue – myClass: Exp(2)

Table 4.8: Features of `gallery_detm1`.

```
source.setArrival(oclass, Det(1))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_detm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: D/M/1
```

4.2.8 `gallery_dm1`

The D/M/1 queue: deterministic interarrival times, exponential service, and a single server.

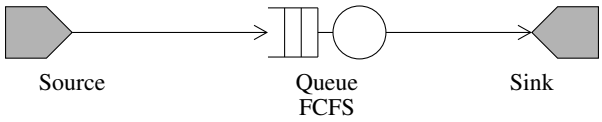

Figure 4.8: Topology of `gallery_dm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Det(1)
Service	Queue – Class1: Exp(2)

Table 4.9: Features of `gallery_dm1`.

The example is built and solved as follows.

```
model = Network('D/M/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Det(1))
queue.setService(oclass1, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_dm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: D/M/1
```

4.2.9 gallery_erldk

The Erl/D/k queue: Erlang interarrival times, deterministic service, and k parallel servers.

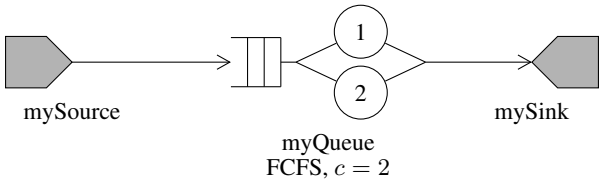

Figure 4.9: Topology of gallery_erldk: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Erlang(5,5)
Service	myQueue – myClass: Det(1)

Table 4.10: Features of gallery_erldk.

The example is built and solved as follows.

```
model = Network('Erl/D/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Erlang.fit_mean_and_order(1, 5))
queue.setService(oclass, Det(2/k))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_erldk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Erl/D/k
```

4.2.10 gallery_erlerl1

The Erl/Erl/1 queue: Erlang interarrival times, Erlang service, and a single server.

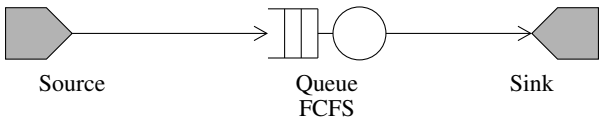

Figure 4.10: Topology of gallery_erlerl1: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('Erl/Erl/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
```

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Erlang(5,5); Class2: Disabled
Service	Queue – Class1: Erlang(10,5); Class2: Exp(3)
Features	class switching on 2 links

Table 4.11: Features of `gallery_erlerl1`.

```
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Erlang.fit_mean_and_order(1, n))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Erlang.fit_mean_and_order(0.5, n))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_erlerl1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Erl/Erl/1
```

4.2.11 `gallery_erlerl1_reentrant`

An Erl/Erl/1 queue with re-entrant visits.

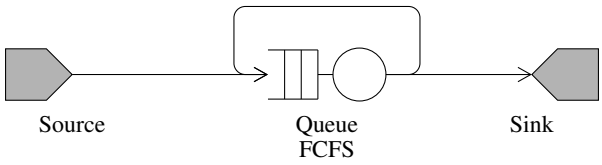

Figure 4.11: Topology of `gallery_erlerl1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Erlang(5,5); Class2: Disabled
Service	Queue – Class1: Erlang(50,5); Class2: Exp(10)
Features	class switching on 1 link

Table 4.12: Features of `gallery_erlerl1_reentrant`.

The example is built and solved as follows.

```
model = Network('Erl/Erl/1-Reentrant')
n = 5
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
```

```
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Erlang.fit_mean_and_order(1, n))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Erlang.fit_mean_and_order(0.1, n))
queue.setService(oclass2, Exp(10))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 0.5)
P.set(oclass2, oclass2, queue, sink, 0.5)
model.link(P)
return model
model = gallery_erlerl1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Erl/Erl/1-Reentrant
```

4.2.12 `gallery_erlm1`

The Erl/M/1 queue: Erlang interarrival times, exponential service, and a single server.

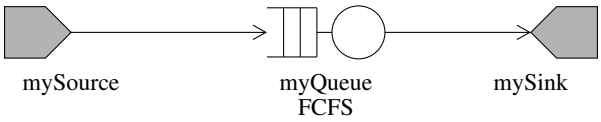

Figure 4.12: Topology of `gallery_erlm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Erlang(5,5)
Service	myQueue – myClass: Exp(2)

Table 4.13: Features of `gallery_erlm1`.

The example is built and solved as follows.

```
model = Network('Er/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Erlang.fit_mean_and_order(1, 5))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_erlm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Er/M/1
```

4.2.13 gallery_erlm1_ps

The Erl/M/1 queue: Erlang interarrival times, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS.

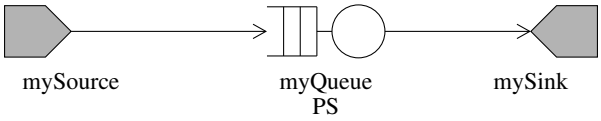

Figure 4.13: Topology of gallery_erlm1_ps: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: PS
Arrivals	mySource – myClass: Erlang(5,5)
Service	myQueue – myClass: Exp(2)

Table 4.14: Features of gallery_erlm1_ps.

The example is built and solved as follows.

```
model = Network('Er/M/1-PS')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.PS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Erlang.fitMeanAndOrder(1, 5))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_erlm1_ps()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Er/M/1-PS
```

4.2.14 gallery_erlm1_reentrant

The Erl/M/1 queue: Erlang interarrival times, exponential service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

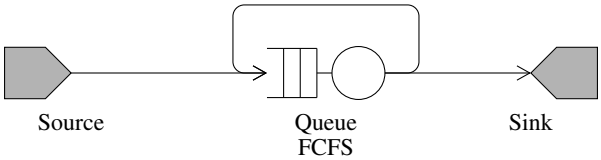

Figure 4.14: Topology of gallery_erlm1_reentrant: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('Er/M/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
```

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Erlang(5,5); Class2: Disabled
Service	Queue – Class1: Exp(2); Class2: Exp(3)
Features	class switching on 1 link

Table 4.15: Features of `gallery_erlm1_reentrant`.

```
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Erlang.fit_mean_and_order(1, 5))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Exp(2))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_erlm1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Er/M/1-Reentrant
```

4.2.15 `gallery_erlm1ps`

The Erl/M/1 queue: Erlang interarrival times, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS.

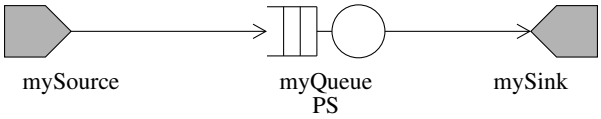

Figure 4.15: Topology of `gallery_erlm1ps`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: PS
Arrivals	mySource – myClass: Erlang(5,5)
Service	myQueue – myClass: Exp(2)

Table 4.16: Features of `gallery_erlm1ps`.

The example is built and solved as follows.

```
model = Network('Er/M/1-PS')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.PS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
```

```
source.setArrival(oclass, Erlang.fit_mean_and_order(1, 5))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_erlm1ps()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Er/M/1-PS
```

4.2.16 gallery_fcr

A finite capacity region, which caps the number of jobs admitted to a set of stations irrespective of their individual buffers.

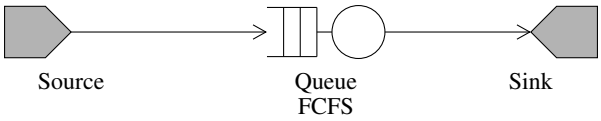

Figure 4.16: Topology of gallery_fcr: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(0.8)
Service	Queue – Class1: Exp(1)

Table 4.17: Features of gallery_fcr.

The example is built and solved as follows.

```
"""Finite capacity region with dropping around a single queue (like M/M/1/K)."""
model = Network('FCR-Dropping')
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Class1', 0)
source.setArrival(oclass, Exp(0.8))
queue.setService(oclass, Exp(1.0))
P = model.init_routing_matrix()
P.set(oclass, oclass, source, queue, 1.0)
P.set(oclass, oclass, queue, sink, 1.0)
model.link(P)
fcr = model.add_region([queue])
fcr.setGlobalMaxJobs(K)
fcr.setDropRule(oclass, True)
return model
model = gallery_fcr()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.17 gallery_fj_closed

The closed fork-join counterpart, with a fixed population circulating through the block.

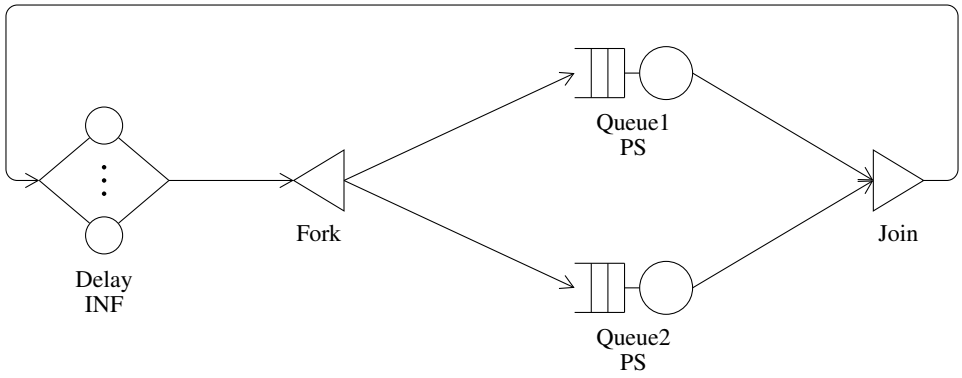

Figure 4.17: Topology of `gallery_fj_closed`: a closed network over 3 queueing stations with a fork-join block.

Nodes	5: Queue × 2, Delay, Fork, Join
Classes	1: class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS
Service	Delay – class1: Exp(1) Queue1 – class1: Exp(1) Queue2 – class1: Exp(1)

Table 4.18: Features of `gallery_fj_closed`.

The example is built and solved as follows.

```
"""Closed fork-join network (single class, two parallel tasks)."""
model = Network('Fork-Join-Closed')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
oclass = ClosedClass(model, 'class1', 5, delay)
delay.setService(oclass, Exp(1.0))
queue1.setService(oclass, Exp(1.0))
queue2.setService(oclass, Exp(1.0))
P = model.init_routing_matrix()
P.set(oclass, oclass, delay, fork, 1.0)
P.set(oclass, oclass, fork, queue1, 1.0)
P.set(oclass, oclass, fork, queue2, 1.0)
P.set(oclass, oclass, queue1, join, 1.0)
P.set(oclass, oclass, queue2, join, 1.0)
P.set(oclass, oclass, join, delay, 1.0)
model.link(P)
return model
model = gallery_fj_closed()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.18 `gallery_fj_open`

An open fork-join block: a job forks into parallel tasks and its siblings are joined before departure.
The example is built and solved as follows.

```
"""Open fork-join network (single class, two parallel tasks)."""
model = Network('Fork-Join-Open')
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
```

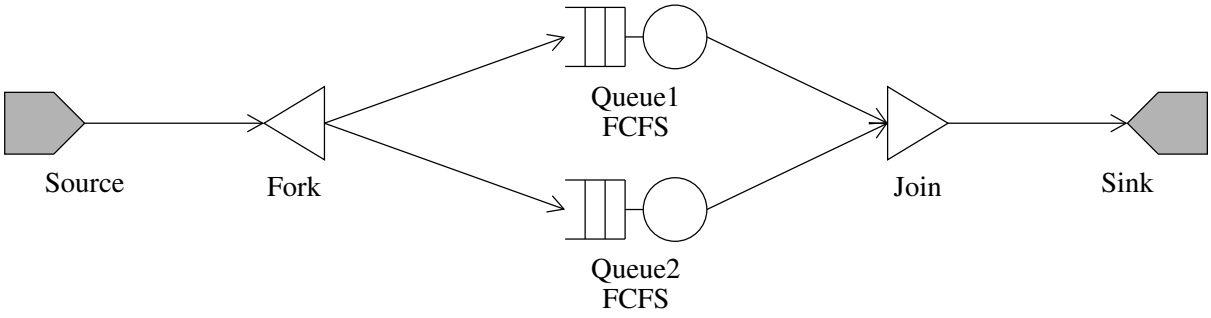

Figure 4.18: Topology of `gallery_fj_open`: an open network over 2 queueing stations with a fork-join block.

Nodes	6: Source, Queue × 2, Fork, Join, Sink
Classes	1: class1 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – class1: Exp(0.05)
Service	Queue1 – class1: Exp(1) Queue2 – class1: Exp(2)

Table 4.19: Features of `gallery_fj_open`.

```
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'class1')
source.setArrival(oclass, Exp(0.05))
queue1.setService(oclass, Exp(1.0))
queue2.setService(oclass, Exp(2.0))
P = model.init_routing_matrix()
P.set(oclass, oclass, source, fork, 1.0)
P.set(oclass, oclass, fork, queue1, 1.0)
P.set(oclass, oclass, fork, queue2, 1.0)
P.set(oclass, oclass, queue1, join, 1.0)
P.set(oclass, oclass, queue2, join, 1.0)
P.set(oclass, oclass, join, sink, 1.0)
model.link(P)
return model
model = gallery_fj_open()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.19 `gallery_fj_quorum`

A fork-join block whose join fires on a quorum of the siblings, the later ones being discarded when they arrive.

Nodes	6: Queue × 3, Delay, Fork, Join
Classes	1: class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: PS; Queue2: PS; Queue3: PS
Service	Delay – class1: Exp(1) Queue1 – class1: Exp(2) Queue2 – class1: Exp(2) Queue3 – class1: Exp(2)
Solvers	LDES

Table 4.20: Features of `gallery_fj_quorum`.

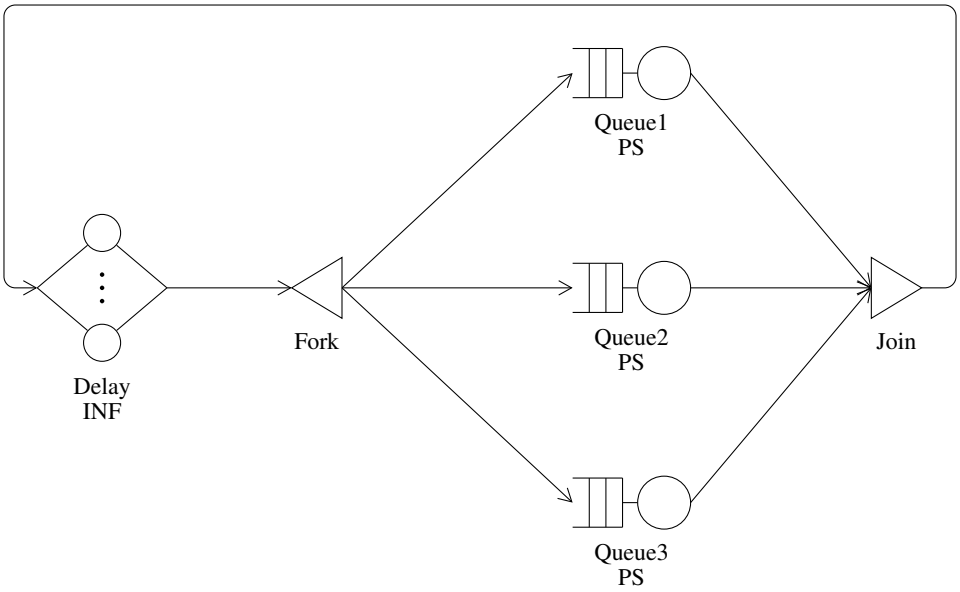

Figure 4.19: Topology of `gallery_fj_quorum`: a closed network over 4 queueing stations with a fork-join block.

The example is built and solved as follows.

```
"""Closed fork-join network with a 2-of-3 quorum join.
The join fires on the SECOND of the three sibling tasks; the third is
discarded when it arrives. Solved exactly by SolverLDES and SolverJMT;
SolverMVA and SolverNC charge the second order statistic of the branch
completion times at the join (fj_ordstat_exp).
"""
model = Network('Fork-Join-Quorum')
delay = Delay(model, 'Delay')
queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
queue3 = Queue(model, 'Queue3', SchedStrategy.PS)
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
oclass = ClosedClass(model, 'class1', 5, delay)
delay.setService(oclass, Exp(1.0))
queue1.setService(oclass, Exp(2.0))
queue2.setService(oclass, Exp(2.0))
queue3.setService(oclass, Exp(2.0))
join.setStrategy(oclass, JoinStrategy.PARTIAL)
join.setRequired(oclass, 2)
P = model.init_routing_matrix()
P.set(oclass, oclass, delay, fork, 1.0)
P.set(oclass, oclass, fork, queue1, 1.0)
P.set(oclass, oclass, fork, queue2, 1.0)
P.set(oclass, oclass, fork, queue3, 1.0)
P.set(oclass, oclass, queue1, join, 1.0)
P.set(oclass, oclass, queue2, join, 1.0)
P.set(oclass, oclass, queue3, join, 1.0)
P.set(oclass, oclass, join, delay, 1.0)
model.link(P)
return model
model = gallery_fj_quorum()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
SolverLDES command: common/ldes solve <tmp> -o <tmp> --seed 23000
  Station JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
0   Delay   class1  1.85748  1.84667  1.00623  1.00623  1.84871  1.84667
```

```
1  Queue1   class1  5.13892  0.92836  2.78086  2.78086  1.84863  1.84675
2  Queue2   class1  4.71060  0.91861  2.54903  2.54903  1.84863  1.84660
3  Queue3   class1  4.96498  0.92577  2.68778  2.68778  1.84863  1.84667
4    Join   class1  2.06165  0.00000  0.55799  0.55799  5.54546  1.84671
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `LDES`.

4.2.20 `gallery_gamm1`

The Gamma/M/1 queue: Gamma interarrival times, exponential service, and a single server.

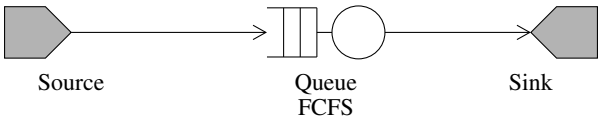

Figure 4.20: Topology of `gallery_gamm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Gamma(5,0.2)
Service	Queue – Class1: Exp(2)

Table 4.21: Features of `gallery_gamm1`.

The example is built and solved as follows.

```
model = Network('Gam/M/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Gamma.fit_mean_and_scv(1, 1/5))
queue.setService(oclass1, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_gamm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Gam/M/1
```

4.2.21 `gallery_hyperl1_feedback`

A HyperExp/Erl/1 queue with probabilistic feedback.

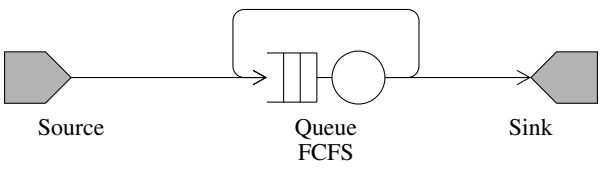

Figure 4.21: Topology of `gallery_hyperl1_feedback`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: HyperExp(0.99,0.01;2.294,0.01759)
Service	Queue – Class1: Erlang(100,5)

Table 4.22: Features of `gallery_hyperl1_feedback`.

The example is built and solved as follows.

```
model = Network('Hyper/Erl/1-Feedback')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, HyperExp.fit_mean_and_scv(1, 64))
queue.setService(oclass1, Erlang.fit_mean_and_order(0.05, 5))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, queue, 0.9)
P.set(oclass1, oclass1, queue, sink, 0.1)
model.link(P)
return model
model = gallery_hyperl1_feedback()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyper/Erl/1-Feedback
```

4.2.22 `gallery_hyperl1_reentrant`

The HyperExp/Erl/1 queue: hyperexponential interarrival times, Erlang service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

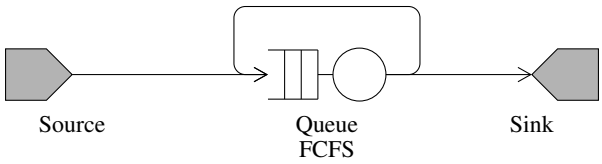

Figure 4.22: Topology of `gallery_hyperl1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: HyperExp(0.99,0.01;2.294,0.01759); Class2: Disabled
Service	Queue – Class1: Erlang(10,5); Class2: Exp(3)
Features	class switching on 1 link

Table 4.23: Features of `gallery_hyperl1_reentrant`.

The example is built and solved as follows.

```
model = Network('Hyper/Erl/1-Reentrant')
```

```
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, HyperExp.fit_mean_and_scv(1, 64))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Erlang.fit_mean_and_order(0.5, 5))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_hyperl1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyper/Erl/1-Reentrant
```

4.2.23 `gallery_hyperlk`

The HyperExp/Erl/k queue: hyperexponential interarrival times, Erlang service, and k parallel servers.

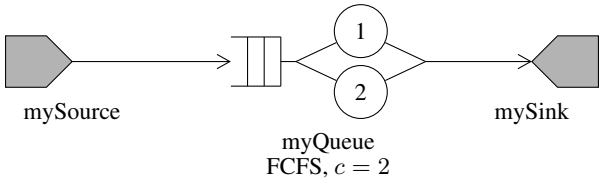

Figure 4.23: Topology of `gallery_hyperlk`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: HyperExp(0.1127,0.8873;0.4057,3.194)
Service	myQueue – myClass: Erlang(4,4)

Table 4.24: Features of `gallery_hyperlk`.

The example is built and solved as follows.

```
model = Network('Hyper/Erl/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, HyperExp.fit_mean_and_scv_balanced(1/1.8, 4))
queue.setService(oclass, Erlang.fit_mean_and_scv(1, 0.25))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_hyperlk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyper/Erl/k
```

4.2.24 gallery_hyphyp1_linear

The HyperExp/HyperExp/1 queue: hyperexponential interarrival times, hyperexponential service, and a single server. In this variant, the model is a linear chain of such stations.

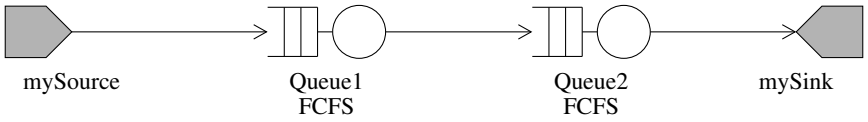

Figure 4.24: Topology of gallery_hyphyp1_linear: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: HyperExp(0.99,0.01;1.077,0.1244)
Service	Queue1 – myClass: HyperExp(0.99,0.01;1.196,0.1383) Queue2 – myClass: HyperExp(0.99,0.01;1.235,0.1015)

Table 4.25: Features of gallery_hyphyp1_linear.

The example is built and solved as follows.

```
model = Network('Hyp/Hyp/1-Linear')
# Block 1: nodes
line = [Source(model, 'mySource')]
for i in range(n):
    line.append(Queue(model, f'Queue{i+1}', SchedStrategy.FCFS))
line.append(Sink(model, 'mySink'))
# Block 2: classes
oclass = OpenClass(model, 'myClass')
line[0].setArrival(oclass, HyperExp.fit_mean_and_scv(1, 2))
means = [Umax] if n//2 == 1 else list(np.linspace(0.1, Umax, n//2))  # MATLAB linspace(a,b,1)
    ==b
if n % 2 == 0:
    means = means + means[::-1]
else:
    means = means + [Umax] + means[::-1]
for i in range(n):
    line[i+1].setService(oclass, HyperExp.fit_mean_and_scv(means[i], 1+i+1))
# Block 3: topology
model.link(Network.serial_routing(line))
return model
model = gallery_hyphyp1_linear()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyp/Hyp/1-Linear
```

4.2.25 gallery_hyphyp1_reentrant

The HyperExp/HyperExp/1 queue: hyperexponential interarrival times, hyperexponential service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.
The example is built and solved as follows.

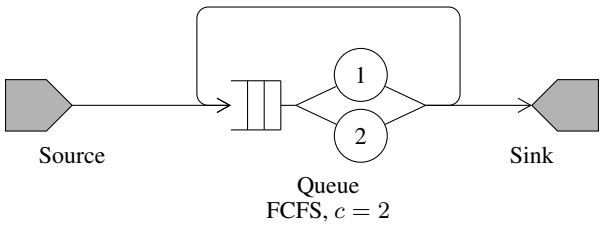

Figure 4.25: Topology of `gallery_hyphyp1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS ($c = 2$)
Arrivals	Source – Class1: HyperExp(0.99,0.01;2.294,0.01759); Class2: Disabled
Service	Queue – Class1: HyperExp(0.99,0.01;2.281,0.1517); Class2: Exp(3)
Features	class switching on 1 link

Table 4.26: Features of `gallery_hyphyp1_reentrant`.

```
model = Network('Hyper/Hyper/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
queue.setNumberOfServers(2)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, HyperExp.fit_mean_and_scv(1, 64))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, HyperExp.fit_mean_and_scv(0.5, 4))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_hyphyp1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyper/Hyper/1-Reentrant
```

4.2.26 `gallery_hyphyp1_tandem`

Two HyperExp/HyperExp/1 queues in series.

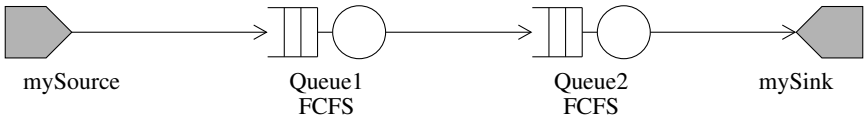

Figure 4.26: Topology of `gallery_hyphyp1_tandem`: an open network over 2 queueing stations.

The example is built and solved as follows.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: HyperExp(0.99,0.01;1.077,0.1244)
Service	Queue1 – myClass: HyperExp(0.99,0.01;1.196,0.1383) Queue2 – myClass: HyperExp(0.99,0.01;1.235,0.1015)

Table 4.27: Features of `gallery_hyphyp1_tandem`.

```
return gallery_hyphyp1_linear(2)
model = gallery_hyphyp1_tandem()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyp/Hyp/1-Linear
```

4.2.27 gallery_hypm1

The HyperExp/M/1 queue: hyperexponential interarrival times, exponential service, and a single server.

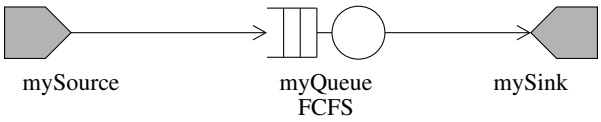

Figure 4.27: Topology of `gallery_hypm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: HyperExp(0.99,0.01;2.294,0.01759)
Service	myQueue – myClass: Exp(2)

Table 4.28: Features of `gallery_hypm1`.

The example is built and solved as follows.

```
model = Network('H2/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, HyperExp.fit_mean_and_scv(1, 64))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_hypm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: H2/M/1
```

4.2.28 gallery_hypm1_reentrant

The HyperExp/M/1 queue: hyperexponential interarrival times, exponential service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

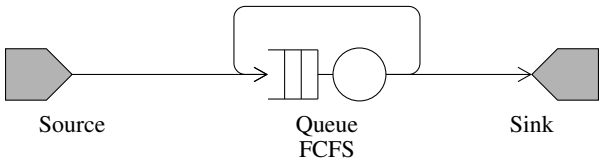

Figure 4.28: Topology of gallery_hypm1_reentrant: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: HyperExp(0.99,0.01;1.14,0.07584); Class2: Disabled
Service	Queue – Class1: Exp(2); Class2: Exp(3)
Features	class switching on 1 link

Table 4.29: Features of gallery_hypm1_reentrant.

The example is built and solved as follows.

```
model = Network('Hyper/M/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, HyperExp.fit_mean_and_scv(1, 4))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Exp(2))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_hypm1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Hyper/M/1-Reentrant
```

4.2.29 gallery_lqn_basic

The smallest layered queueing network: one reference task calling one server task.
The example is built and solved as follows.

```
"""Basic 3-task layered queueing network (client/server chain)."""
model = LayeredNetwork('LQN-Basic')
P1 = Processor(model, 'P1', 2, SchedStrategy.PS)
P2 = Processor(model, 'P2', 3, SchedStrategy.PS)
T1 = Task(model, 'T1', 50, SchedStrategy.REF).on(P1).set_think_time(Exp(1 / 2))
T2 = Task(model, 'T2', 50, SchedStrategy.FCFS).on(P1).set_think_time(Exp(1 / 3))
```

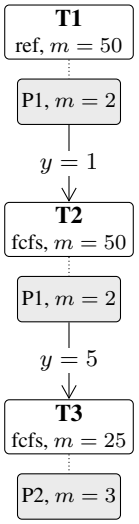

Figure 4.29: Call graph of `gallery_lqn_basic`: 3 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	2: P1 (ps, $m = 2$), P2 (ps, $m = 3$)
Tasks	3: T1 (ref, $m = 50$), T2 (fcfs, $m = 50$), T3 (fcfs, $m = 25$)
Entries	3: E1, E2, E3
Activities	3, host demands AS1: 0.1, AS2: 0.05, AS3: 0.02
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 2.0

Table 4.30: Features of `gallery_lqn_basic`.

```
T3 = Task(model, 'T3', 25, SchedStrategy.FCFS).on(P2).set_think_time(Exp(1 / 4))
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
E3 = Entry(model, 'E3').on(T3)
A1 = Activity(model, 'AS1', Exp(10)).on(T1).bound_to(E1).synch_call(E2, 1)
A2 = Activity(model, 'AS2', Exp(20)).on(T2).bound_to(E2).synch_call(E3, 5).replies_to(E2)
A3 = Activity(model, 'AS3', Exp(50)).on(T3).bound_to(E3).replies_to(E3)
return model
model = gallery_lqn_basic()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: LayeredNetwork
```

4.2.30 `gallery_lqn_random`

A randomly generated layered model, used to exercise the layered solvers beyond hand-written topologies.

Processors	3: c_processor_1 (inf, $m = 1$), processor_1 (ps, $m = 1$), processor_2 (ps, $m = 1$)
Tasks	5: c_task_1 (ref, $m = 1$), task_1 (fcfs, $m = 1$), task_2 (fcfs, $m = 1$), task_3 (fcfs, $m = 1$), task_4 (fcfs, $m = 1$)
Entries	5: c_entry_1, entry_1, entry_2, entry_3, entry_4
Activities	5, host demands c_activity_1: 1.0, activity_1: 1.0, activity_2: 1.0, activity_3: 1.0, activity_4: 1.0
Calls	4 (4 synchronous, 0 asynchronous)

Table 4.31: Features of `gallery_lqn_random`.

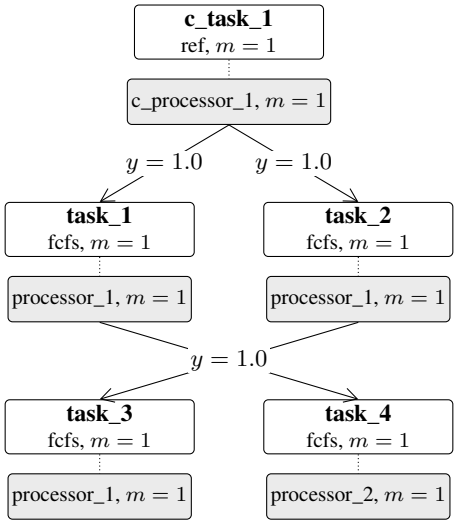

Figure 4.30: Call graph of `gallery_lqn_random`: 5 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
"""Randomly generated layered queueing network (reproducible).
Uses LayeredNetworkGenerator with a fixed default seed so repeated calls
yield the same model. 1 client, 2 levels, 4 tasks, 2 processors.
"""
random.seed(seed)
np.random.seed(seed)
gen = LayeredNetworkGenerator()
return gen.generate(1, 2, 4, 2)
model = gallery_lqn_random()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: LayeredNetwork
```

4.2.31 `gallery_lqn_workflows`

Workflow patterns (sequence, AND-fork and join, loop) as an activity graph in a layered model.

Processors	3: P1 (inf, $m = 1$), P2 (inf, $m = 1$), P3 (ps, $m = 5$)
Tasks	3: T1 (ref, $m = 1$), T2 (inf, $m = 1$), T3 (inf, $m = 1$)
Entries	3: Entry, E2, E3
Activities	14, host demands A1: 1.0, A2: 2.0, A3: 3.0, B1: 0.1, B2: 0.2, B3: 0.3, B4: 0.4, B5: 0.5, ...
Calls	2 (2 synchronous, 0 asynchronous)
Think times	T1: 0.0

Table 4.32: Features of `gallery_lqn_workflows`.

The example is built and solved as follows.

```
"""Layered network with loop, and-fork/join and or-fork/join precedence."""
model = LayeredNetwork('LQN-Workflows')
P1 = Processor(model, 'P1', float('inf'), SchedStrategy.INF)
T1 = Task(model, 'T1', 1, SchedStrategy.REF).on(P1)
T1.set_think_time(Immediate())
E1 = Entry(model, 'Entry').on(T1)
```

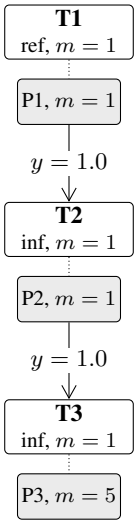

Figure 4.31: Call graph of `gallery_lqn_workflows`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
P2 = Processor(model, 'P2', float('inf'), SchedStrategy.INF)
T2 = Task(model, 'T2', float('inf'), SchedStrategy.INF).on(P2).set_think_time(Immediate())
E2 = Entry(model, 'E2').on(T2)
P3 = Processor(model, 'P3', 5, SchedStrategy.PS)
T3 = Task(model, 'T3', float('inf'), SchedStrategy.INF).on(P3)
T3.set_think_time(Exp.fit_mean(10))
E3 = Entry(model, 'E3').on(T3)
A1 = Activity(model, 'A1', Exp.fit_mean(1)).on(T1).bound_to(E1)
A2 = Activity(model, 'A2', Exp.fit_mean(2)).on(T1)
A3 = Activity(model, 'A3', Exp.fit_mean(3)).on(T1).synch_call(E2)
B1 = Activity(model, 'B1', Exp.fit_mean(0.1)).on(T2).bound_to(E2)
B2 = Activity(model, 'B2', Exp.fit_mean(0.2)).on(T2)
B3 = Activity(model, 'B3', Exp.fit_mean(0.3)).on(T2)
B4 = Activity(model, 'B4', Exp.fit_mean(0.4)).on(T2)
B5 = Activity(model, 'B5', Exp.fit_mean(0.5)).on(T2)
B6 = Activity(model, 'B6', Exp.fit_mean(0.6)).on(T2).synch_call(E3).replies_to(E2)
C1 = Activity(model, 'C1', Exp.fit_mean(0.1)).on(T3).bound_to(E3)
C2 = Activity(model, 'C2', Exp.fit_mean(0.2)).on(T3)
C3 = Activity(model, 'C3', Exp.fit_mean(0.3)).on(T3)
C4 = Activity(model, 'C4', Exp.fit_mean(0.4)).on(T3)

    ... (4 source lines omitted; full implementation: python/examples/gallery/)

T2.add_precedence(ActivityPrecedence.AndJoin([B2, B3, B5], B6))
T3.add_precedence(ActivityPrecedence.OrFork(C1, [C2, C3, C4], [0.3, 0.3, 0.4]))
T3.add_precedence(ActivityPrecedence.OrJoin([C2, C3, C4], C5))
return model
model = gallery_lqn_workflows()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: LayeredNetwork
```

4.2.32 `gallery_lukumar_reentrant`

The Lu-Kumar re-entrant line, the standard counterexample in which a work-conserving policy destabilizes a network whose stations are individually underloaded.
The example is built and solved as follows.

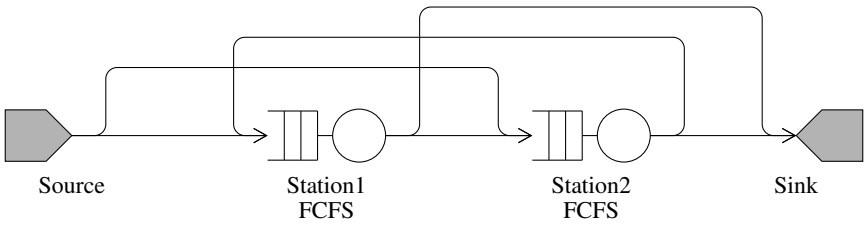

Figure 4.32: Topology of `gallery_lukumar_reentrant`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	4: Class1 (open), Class2 (open), Class3 (open), Class4 (open)
Scheduling	Station1: FCFS; Station2: FCFS
Arrivals	Source – Class1: Exp(0.08); Class2: Disabled; Class3: Exp(0.08); Class4: Disabled
Service	Station1 – Class1: Exp(0.1); Class2: Disabled; Class3: Disabled; Class4: Exp(1) Station2 – Class1: Disabled; Class2: Exp(1); Class3: Exp(0.1); Class4: Disabled
Features	class switching on 1 link

Table 4.33: Features of `gallery_lukumar_reentrant`.

```
# Convert string to scheduling strategy
sched_strategy = sched_strategy.upper()
if sched_strategy == 'FCFS':
    sched1 = SchedStrategy.FCFS
    sched2 = SchedStrategy.FCFS
elif sched_strategy == 'HOL':
    sched1 = SchedStrategy.HOL
    sched2 = SchedStrategy.HOL
elif sched_strategy == 'PS':
    sched1 = SchedStrategy.PS
    sched2 = SchedStrategy.PS
else:
    sched1 = SchedStrategy.FCFS
    sched2 = SchedStrategy.FCFS
model = Network('Lu-Kumar-Reentrant')
# Block 1: Nodes
source = Source(model, 'Source')
station1 = Queue(model, 'Station1', sched1)
station2 = Queue(model, 'Station2', sched2)
sink = Sink(model, 'Sink')
# Block 2: Job Classes
# For HOL scheduling, priority is specified as the third parameter
# Higher value = higher priority
#
# "Bad" priority policy for instability demonstration (FBFS):
#   Station 1: Class 1 (priority 1) > Class 4 (priority 0)

    ... (59 source lines omitted; full implementation: python/examples/gallery/)

P.set(class3, class4, station2, station1, 1)    # Class3@Station2 -> Class4@Station1
P.set(class4, class4, station1, sink, 1)        # Class4@Station1 -> Sink
model.link(P)
return model
model = gallery_lukumar_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Lu-Kumar-Reentrant
```

4.2.33 gallery_mapm1

The MAP/M/1 queue: a Markovian arrival process, exponential service, and a single server.

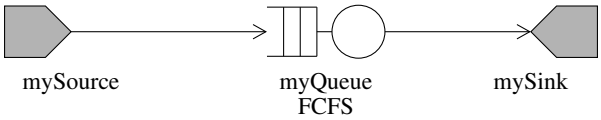

Figure 4.33: Topology of gallery_mapm1: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: MAP
Service	myQueue – myClass: Exp(2)

Table 4.34: Features of gallery_mapm1.

The example is built and solved as follows.

```
if map is None:
    D0 = np.array([[-0.6984901916396979, 0.45234650636128054],
                   [0.34690024319398277, -0.8194057961021199]])
    D1 = np.array([[0.2125067546435463, 0.033636930634871165],
                   [0.4441520099524867, 0.028353542955650513]])
    map = MAP(D0, D1)
model = Network('MAP/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, map)
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mapm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: MAP/M/1
```

4.2.34 gallery_mapmk

The MAP/M/k queue: a Markovian arrival process, exponential service, and k parallel servers.

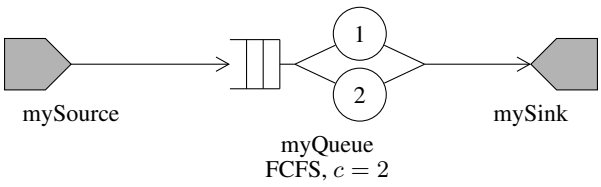

Figure 4.34: Topology of gallery_mapmk: an open network over 1 queueing station.

The example is built and solved as follows.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: MAP
Service	myQueue – myClass: Exp(2)

Table 4.35: Features of `gallery_mapmk`.

```
if map is None:
    map = MAP.rand(seed=seed)
model = Network('MAP/M/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, map)
queue.setService(oclass, Exp(2))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mapmk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: MAP/M/k
```

4.2.35 `gallery_mdk`

The M/D/k queue: Poisson arrivals, deterministic service, and k parallel servers.

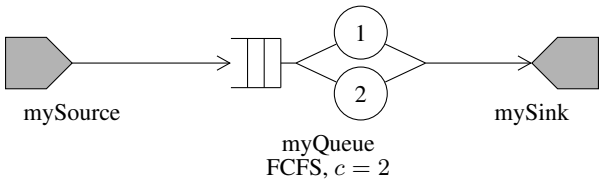

Figure 4.35: Topology of `gallery_mdk`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Det(1)

Table 4.36: Features of `gallery_mdk`.

The example is built and solved as follows.

```
model = Network('M/D/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
```

```
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp.fit_mean(1))
queue.setService(oclass, Det(2/k))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mdk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/D/k
```

4.2.36 gallery_merl1

The M/Erl/1 queue: Poisson arrivals, Erlang service, and a single server.

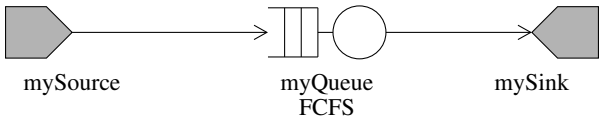

Figure 4.36: Topology of gallery_merl1: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Erlang(4,2)

Table 4.37: Features of gallery_merl1.

The example is built and solved as follows.

```
model = Network('M/E/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Erlang.fit_mean_and_order(0.5, 2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_merl1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/E/1
```

4.2.37 gallery_merl1_linear

The M/Erl/1 queue: Poisson arrivals, Erlang service, and a single server. In this variant, the model is a linear chain of such stations.
The example is built and solved as follows.

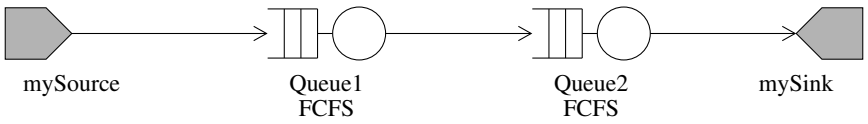

Figure 4.37: Topology of `gallery_merl1_linear`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: Erlang(1.111,1) Queue2 – myClass: Erlang(2.222,2)

Table 4.38: Features of `gallery_merl1_linear`.

```
model = Network('M/Erl/1-Linear')
# Block 1: nodes
line = [Source(model, 'mySource')]
for i in range(n):
    line.append(Queue(model, f'Queue{i+1}', SchedStrategy.FCFS))
line.append(Sink(model, 'mySink'))
# Block 2: classes
oclass = OpenClass(model, 'myClass')
line[0].setArrival(oclass, Exp(1))
means = [Umax] if n//2 == 1 else list(np.linspace(0.1, Umax, n//2))  # MATLAB linspace(a,b,1)
    ==b
if n % 2 == 0:
    means = means + means[::-1]
else:
    means = means + [Umax] + means[::-1]
for i in range(n):
    line[i+1].setService(oclass, Erlang.fit_mean_and_order(means[i], i+1))
# Block 3: topology
model.link(Network.serial_routing(line))
return model
model = gallery_merl1_linear()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Erl/1-Linear
```

4.2.38 `gallery_merl1_reentrant`

The M/Erl/1 queue: Poisson arrivals, Erlang service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

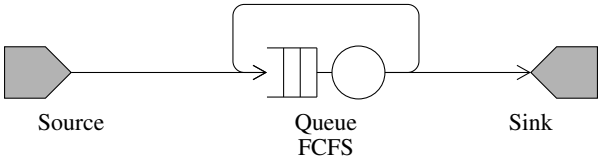

Figure 4.38: Topology of `gallery_merl1_reentrant`: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('M/Erl/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
```

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Disabled
Service	Queue – Class1: Erlang(10,5); Class2: Exp(3)
Features	class switching on 1 link

Table 4.39: Features of `gallery_merl1_reentrant`.

```
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Exp(1))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Erlang.fit_mean_and_order(1/2, 5))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_merl1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Erl/1-Reentrant
```

4.2.39 `gallery_merl1_tandem`

Two M/Erl/1 queues in series.

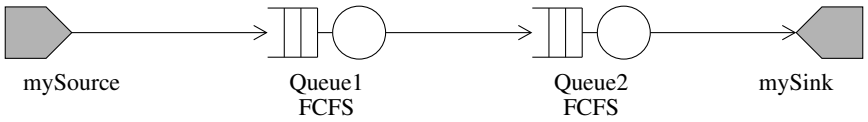

Figure 4.39: Topology of `gallery_merl1_tandem`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: Erlang(1.111,1) Queue2 – myClass: Erlang(2.222,2)

Table 4.40: Features of `gallery_merl1_tandem`.

The example is built and solved as follows.

```
return gallery_merl1_linear(2)
model = gallery_merl1_tandem()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Erl/1-Linear
```

4.2.40 gallery_merlk

The M/Erl/k queue: Poisson arrivals, Erlang service, and k parallel servers.

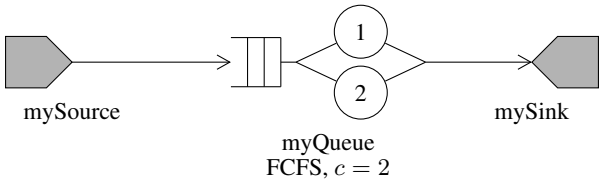

Figure 4.40: Topology of `gallery_merlk`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Erlang(4,2)

Table 4.41: Features of `gallery_merlk`.

The example is built and solved as follows.

```
model = Network(f'M/Erl/{k}')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Erlang.fit_mean_and_order(0.5, 2))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_merlk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Erl/2
```

4.2.41 gallery_mhyp1

The M/HyperExp/1 queue: Poisson arrivals, hyperexponential service, and a single server.

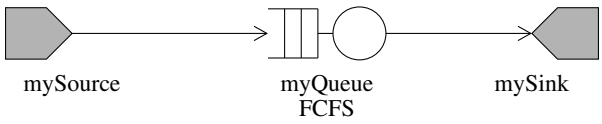

Figure 4.41: Topology of `gallery_mhyp1`: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('M/H/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
```

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: HyperExp(0.99,0.01;2.281,0.1517)

Table 4.42: Features of `gallery_mhyp1`.

```
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, HyperExp.fit_mean_and_scv(0.5, 4))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mhyp1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/H/1
```

4.2.42 `gallery_mhyp1_linear`

The M/HyperExp/1 queue: Poisson arrivals, hyperexponential service, and a single server. In this variant, the model is a linear chain of such stations.

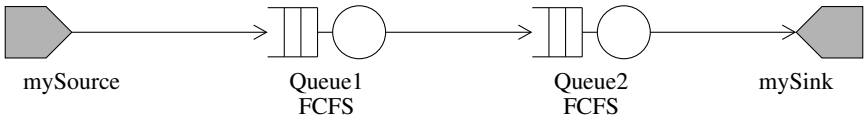

Figure 4.42: Topology of `gallery_mhyp1_linear`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: HyperExp(0.99,0.01;1.196,0.1383) Queue2 – myClass: HyperExp(0.99,0.01;1.196,0.1383)

Table 4.43: Features of `gallery_mhyp1_linear`.

The example is built and solved as follows.

```
model = Network('M/Hyp/1-Linear')
# Block 1: nodes
line = [Source(model, 'mySource')]
for i in range(n):
    line.append(Queue(model, f'Queue{i+1}', SchedStrategy.FCFS))
line.append(Sink(model, 'mySink'))
# Block 2: classes
oclass = OpenClass(model, 'myClass')
line[0].setArrival(oclass, Exp(1))
means = [Umax] if n//2 == 1 else list(np.linspace(0.1, Umax, n//2))  # MATLAB linspace(a,b,1)
    ==b
if n % 2 == 0:
    means = means + means[::-1]
else:
```

```
    means = means + [Umax] + means[::-1]
for i in range(n):
    line[i+1].setService(oclass, HyperExp.fit_mean_and_scv(means[i], n))
# Block 3: topology
model.link(Network.serial_routing(line))
return model
model = gallery_mhyp1_linear()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Hyp/1-Linear
```

4.2.43 `gallery_mhyp1_reentrant`

The M/HyperExp/1 queue: Poisson arrivals, hyperexponential service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

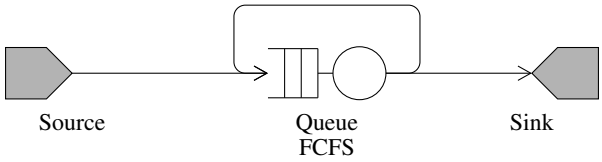

Figure 4.43: Topology of `gallery_mhyp1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Disabled
Service	Queue – Class1: Coxian; Class2: Exp(3)
Features	class switching on 1 link

Table 4.44: Features of `gallery_mhyp1_reentrant`.

The example is built and solved as follows.

```
model = Network('M/Hyper/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Exp(1))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Coxian.fit_mean_and_scv(0.5, 4))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_mhyp1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Hyper/1-Reentrant
```

4.2.44 gallery_mhyp1_tandem

Two M/HyperExp/1 queues in series.

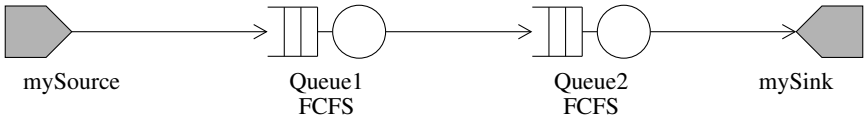

Figure 4.44: Topology of gallery_mhyp1_tandem: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: HyperExp(0.99,0.01;1.196,0.1383) Queue2 – myClass: HyperExp(0.99,0.01;1.196,0.1383)

Table 4.45: Features of gallery_mhyp1_tandem.

The example is built and solved as follows.

```
return gallery_mhyp1_linear(2)
model = gallery_mhyp1_tandem()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Hyp/1-Linear
```

4.2.45 gallery_mhypk

The M/HyperExp/k queue: Poisson arrivals, hyperexponential service, and k parallel servers.

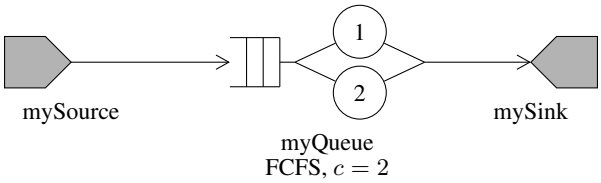

Figure 4.45: Topology of gallery_mhypk: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network(f'M/Hyper/{k}')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Coxian.fit_mean_and_scv(0.5, 4))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mhypk()
print(f"Model: {model.getName()}")
```

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Coxian

Table 4.46: Features of `gallery_mhypk`.

Running this edition prints

```
Model: M/Hyper/2
```

4.2.46 gallery_mm1

The M/M/1 queue: Poisson arrivals, exponential service, and a single server.

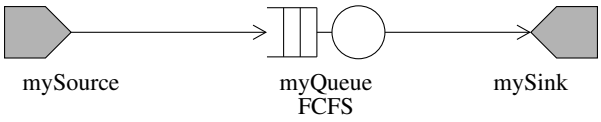

Figure 4.46: Topology of `gallery_mm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Exp(2)

Table 4.47: Features of `gallery_mm1`.

The example is built and solved as follows.

```
model = Network('M/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1
```

4.2.47 gallery_mm1_feedback

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, a completing job returns to the station with fixed probability.
The example is built and solved as follows.

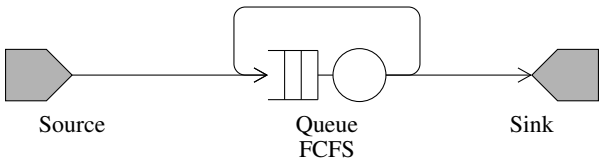

Figure 4.47: Topology of `gallery_mm1_feedback`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Queue – Class1: Exp(2)

Table 4.48: Features of `gallery_mm1_feedback`.

```
model = Network('M/M/1-Feedback')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Exp.fit_mean(1))
queue.setService(oclass1, Exp.fit_mean(0.5))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, queue, p)
P.set(oclass1, oclass1, queue, sink, 1-p)
model.link(P)
return model
model = gallery_mm1_feedback()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-Feedback
```

4.2.48 `gallery_mm1_linear`

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the model is a linear chain of such stations.

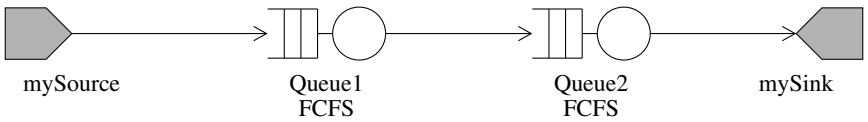

Figure 4.48: Topology of `gallery_mm1_linear`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
model = Network('M/M/1-Linear')
# Block 1: nodes
line = [Source(model, 'mySource')]
for i in range(n):
    line.append(Queue(model, f'Queue{i+1}', SchedStrategy.FCFS))
line.append(Sink(model, 'mySink'))
# Block 2: classes
```

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: Exp(1.111) Queue2 – myClass: Exp(1.111)

Table 4.49: Features of `gallery_mm1_linear`.

```
oclass = OpenClass(model, 'myClass')
line[0].setArrival(oclass, Exp(1))
means = [Umax] if n//2 == 1 else list(np.linspace(0.1, Umax, n//2))  # MATLAB linspace(a,b,1)
    ==b
if n % 2 == 0:
    means = means + means[::-1]
else:
    means = means + [Umax] + means[::-1]
for i in range(n):
    line[i+1].setService(oclass, Exp.fit_mean(means[i]))
# Block 3: topology
model.link(Network.serial_routing(line))
return model
model = gallery_mm1_linear()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-Linear
```

4.2.49 `gallery_mm1_multiclass`

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, several open classes share the station.

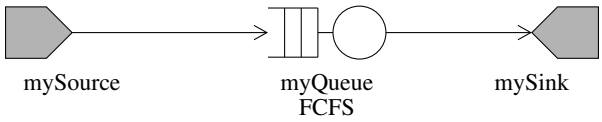

Figure 4.49: Topology of `gallery_mm1_multiclass`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass1: Exp(1); myClass2: Exp(0.5)
Service	myQueue – myClass1: Exp(4); myClass2: Exp(4)

Table 4.50: Features of `gallery_mm1_multiclass`.

The example is built and solved as follows.

```
model = Network('M[2]/M[2]/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(1))
queue.setService(oclass1, Exp(4))
oclass2 = OpenClass(model, 'myClass2')
```

```
source.setArrival(oclass2, Exp(0.5))
queue.setService(oclass2, Exp(4))
# Block 3: topology
P = model.init_routing_matrix()
P[oclass1] = Network.serial_routing([source, queue, sink])
P[oclass2] = Network.serial_routing([source, queue, sink])
model.link(P)
return model
model = gallery_mm1_multiclass()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M[2]/M[2]/1
```

4.2.50 gallery_mm1_prio

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the classes carry priorities, served head-of-line.

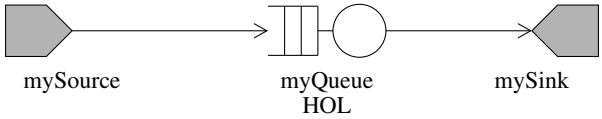

Figure 4.50: Topology of gallery_mm1_prio: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: HOL
Arrivals	mySource – myClass1: Exp(1); myClass2: Exp(0.5)
Service	myQueue – myClass1: Exp(4); myClass2: Exp(4)

Table 4.51: Features of gallery_mm1_prio.

The example is built and solved as follows.

```
model = Network('M[2]/M[2]/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.HOL)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1', 1)
source.setArrival(oclass1, Exp(1))
queue.setService(oclass1, Exp(4))
oclass2 = OpenClass(model, 'myClass2', 0)
source.setArrival(oclass2, Exp(0.5))
queue.setService(oclass2, Exp(4))
# Block 3: topology
P = model.init_routing_matrix()
P[oclass1] = Network.serial_routing([source, queue, sink])
P[oclass2] = Network.serial_routing([source, queue, sink])
model.link(P)
return model
model = gallery_mm1_prio()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M[2]/M[2]/1
```

4.2.51 gallery_mm1_ps

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS.

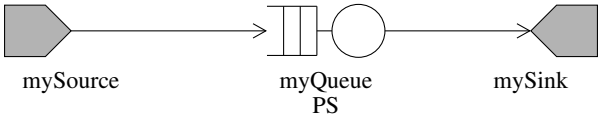

Figure 4.51: Topology of gallery_mm1_ps: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: PS
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Exp(2)

Table 4.52: Features of gallery_mm1_ps.

The example is built and solved as follows.

```
model = Network('M/M/1-PS')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.PS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Exp(2))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mm1_ps()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-PS
```

4.2.52 gallery_mm1_ps_feedback

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS and a completing job returns to the station with fixed probability.

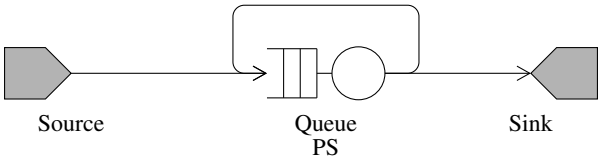

Figure 4.52: Topology of gallery_mm1_ps_feedback: an open network over 1 queueing station.

The example is built and solved as follows.

```
model = Network('M/M/1-PS-Feedback')
# Block 1: nodes
```

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: PS
Arrivals	Source – Class1: Exp(1)
Service	Queue – Class1: Exp(2)

Table 4.53: Features of `gallery_mm1_ps_feedback`.

```
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.PS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Exp.fit_mean(1))
queue.setService(oclass1, Exp.fit_mean(0.5))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, queue, p)
P.set(oclass1, oclass1, queue, sink, 1-p)
model.link(P)
return model
model = gallery_mm1_ps_feedback()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-PS-Feedback
```

4.2.53 `gallery_mm1_ps_multiclass`

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS and several open classes share the station.

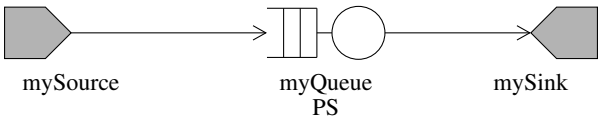

Figure 4.53: Topology of `gallery_mm1_ps_multiclass`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: PS
Arrivals	mySource – myClass1: Exp(1); myClass2: Exp(0.5)
Service	myQueue – myClass1: Exp(4); myClass2: Exp(4)

Table 4.54: Features of `gallery_mm1_ps_multiclass`.

The example is built and solved as follows.

```
model = Network('M[2]/M[2]/1-PS')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.PS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(1))
```

```
queue.setService(oclass1, Exp(4))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.5))
queue.setService(oclass2, Exp(4))
# Block 3: topology
P = model.init_routing_matrix()
P[oclass1] = Network.serial_routing([source, queue, sink])
P[oclass2] = Network.serial_routing([source, queue, sink])
model.link(P)
return model
model = gallery_mm1_ps_multiclass()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M[2]/M[2]/1-PS
```

4.2.54 gallery_mm1_ps_reentrant

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, the station serves under processor sharing rather than FCFS and jobs re-enter the station, so a visit is counted more than once.

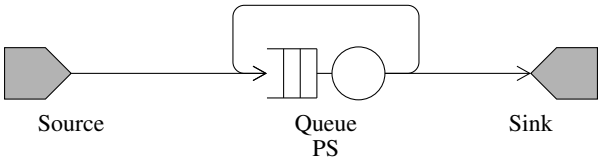

Figure 4.54: Topology of `gallery_mm1_ps_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: PS
Arrivals	Source – Class1: Exp(1); Class2: Disabled
Service	Queue – Class1: Exp(2); Class2: Exp(3)
Features	class switching on 1 link

Table 4.55: Features of `gallery_mm1_ps_reentrant`.

The example is built and solved as follows.

```
model = Network('M/M/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.PS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Exp(1))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Exp(2))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_mm1_ps_reentrant()
```

```
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1
```

4.2.55 gallery_mm1_reentrant

The M/M/1 queue: Poisson arrivals, exponential service, and a single server. In this variant, jobs re-enter the station, so a visit is counted more than once.

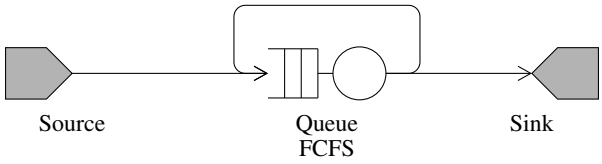

Figure 4.55: Topology of `gallery_mm1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Disabled
Service	Queue – Class1: Exp(2); Class2: Exp(3)
Features	class switching on 1 link

Table 4.56: Features of `gallery_mm1_reentrant`.

The example is built and solved as follows.

```
model = Network('M/M/1-Reentrant')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
oclass2 = OpenClass(model, 'Class2')
source.setArrival(oclass1, Exp(1))
source.setArrival(oclass2, Disabled())
queue.setService(oclass1, Exp(2))
queue.setService(oclass2, Exp(3))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass2, queue, queue, 1)
P.set(oclass2, oclass2, queue, sink, 1)
model.link(P)
return model
model = gallery_mm1_reentrant()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-Reentrant
```

4.2.56 gallery_mm1_tandem

Two M/M/1 queues in series.
The example is built and solved as follows.

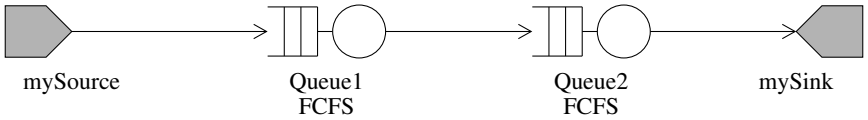

Figure 4.56: Topology of `gallery_mm1_tandem`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: myClass (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	Queue1 – myClass: Exp(1.111) Queue2 – myClass: Exp(1.111)

Table 4.57: Features of `gallery_mm1_tandem`.

```
return gallery_mm1_linear(2)
model = gallery_mm1_tandem()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/1-Linear
```

4.2.57 `gallery_mm1_tandem_multiclass`

Two M/M/1 queues in series under a multiclass workload.

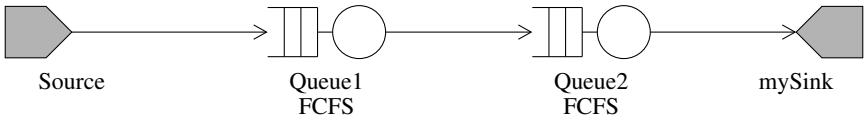

Figure 4.57: Topology of `gallery_mm1_tandem_multiclass`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
model = Network('M[2]/M[2]/1 -> -/M[2]/1')
# Block 1: nodes
source = Source(model, 'Source')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(1))
queue1.setService(oclass1, Exp(4))
queue2.setService(oclass1, Exp(6))
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.5))
queue1.setService(oclass2, Exp(2))
queue2.setService(oclass2, Exp(6))
# Block 3: topology
P = model.init_routing_matrix()
P[oclass1] = Network.serial_routing([source, queue1, queue2, sink])
P[oclass2] = Network.serial_routing([source, queue1, queue2, sink])
model.link(P)
return model
model = gallery_mm1_tandem_multiclass()
print(f"Model: {model.getName()}")
```

Nodes	4: Source, Queue × 2, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – myClass1: Exp(1); myClass2: Exp(0.5)
Service	Queue1 – myClass1: Exp(4); myClass2: Exp(2) Queue2 – myClass1: Exp(6); myClass2: Exp(6)

Table 4.58: Features of `gallery_mm1_tandem_multiclass`.

Running this edition prints

```
Model: M[2]/M[2]/1 -> -/M[2]/1
```

4.2.58 `gallery_mm1k`

The M/M/1/K queue: a single exponential server behind a buffer of K places, so an arrival finding the buffer full is lost.

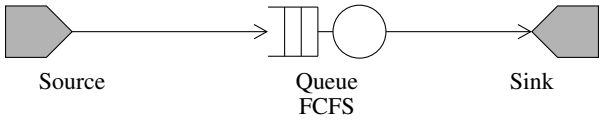

Figure 4.58: Topology of `gallery_mm1k`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(0.8)
Service	Queue – Class1: Exp(1)
Features	finite buffer 3 at Queue

Table 4.59: Features of `gallery_mm1k`.

The example is built and solved as follows.

```
"""M/M/1/K queue with finite capacity K (blocking / loss)."""
model = Network('M/M/1/K')
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
queue.setNumberOfServers(1)
queue.setCapacity(K)
sink = Sink(model, 'Sink')
oclass = OpenClass(model, 'Class1', 0)
source.setArrival(oclass, Exp(0.8))
queue.setService(oclass, Exp(1.0))
P = model.init_routing_matrix()
P.set(oclass, oclass, source, queue, 1.0)
P.set(oclass, oclass, queue, sink, 1.0)
model.link(P)
return model
model = gallery_mm1k()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.59 gallery_mmap1

The M/MAP/1 queue: Poisson arrivals, MAP-modulated service, and a single server.

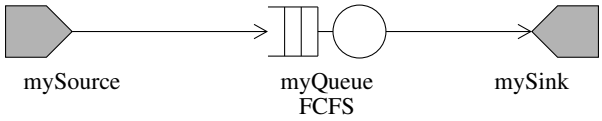

Figure 4.59: Topology of gallery_mmap1: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: MAP

Table 4.60: Features of gallery_mmap1.

The example is built and solved as follows.

```
if map is None:
    map = MAP.rand(seed=seed)
    map = map.set_mean(0.5)
model = Network('M/MAP/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, map)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mmap1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/MAP/1
```

4.2.60 gallery_mmap1_multiclass

The M/MAP/1 queue: Poisson arrivals, MAP-modulated service, and a single server. In this variant, several open classes share the station.

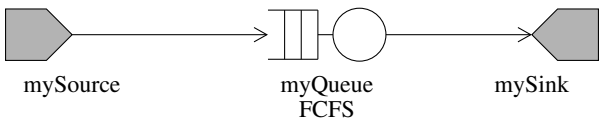

Figure 4.60: Topology of gallery_mmap1_multiclass: an open network over 1 queueing station.

The example is built and solved as follows.

```
if map1 is None:
    map1 = MAP.rand(n=2, seed=seed)
    map1 = map1.set_mean(0.5)
if map2 is None:
```

Nodes	3: Source, Queue, Sink
Classes	2: myClass1 (open), myClass2 (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass1: Exp(0.7); myClass2: Exp(0.3)
Service	myQueue – myClass1: MAP; myClass2: MAP

Table 4.61: Features of `gallery_mmap1_multiclass`.

```
    map2 = MAP.rand(n=3, seed=seed+1)
    map2 = map2.set_mean(0.5)
model = Network('M/MAP/1')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass1 = OpenClass(model, 'myClass1')
source.setArrival(oclass1, Exp(0.35 / map1.get_mean()))
queue.setService(oclass1, map1)
oclass2 = OpenClass(model, 'myClass2')
source.setArrival(oclass2, Exp(0.15 / map2.get_mean()))
queue.setService(oclass2, map2)
# Block 3: topology
P = model.init_routing_matrix()
P[oclass1] = Network.serial_routing([source, queue, sink])
P[oclass2] = Network.serial_routing([source, queue, sink])
model.link(P)
return model
model = gallery_mmap1_multiclass()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/MAP/1
```

4.2.61 `gallery_mmapk`

The M/MAP/k queue: Poisson arrivals, MAP-modulated service, and k parallel servers.

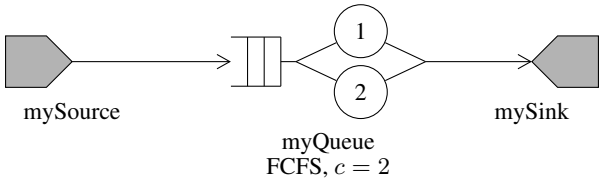

Figure 4.61: Topology of `gallery_mmapk`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: MAP

Table 4.62: Features of `gallery_mmapk`.

The example is built and solved as follows.

```
if map is None:
    map = MAP.rand(seed=seed)
```

```
model = Network('M/MAP/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, map)
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mmapk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/MAP/k
```

4.2.62 `gallery_mmk`

The M/M/k queue: Poisson arrivals, exponential service, and k parallel servers.

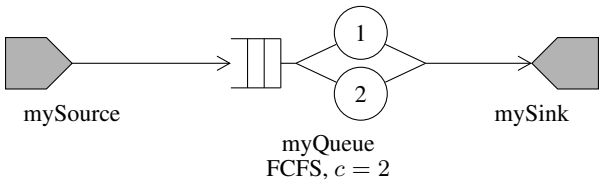

Figure 4.62: Topology of `gallery_mmk`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS ($c = 2$)
Arrivals	mySource – myClass: Exp(1)
Service	myQueue – myClass: Exp(2)

Table 4.63: Features of `gallery_mmk`.

The example is built and solved as follows.

```
model = Network('M/M/k')
# Block 1: nodes
source = Source(model, 'mySource')
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
source.setArrival(oclass, Exp(1))
queue.setService(oclass, Exp(2))
queue.setNumberOfServers(k)
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_mmk()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/M/k
```

4.2.63 gallery_mpar1

The M/Pareto/1 queue: Poisson arrivals, Pareto service, and a single server.

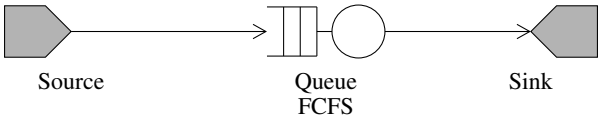

Figure 4.63: Topology of gallery_mpar1: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Queue – Class1: Pareto(2.008,0.251)

Table 4.64: Features of gallery_mpar1.

The example is built and solved as follows.

```
model = Network('M/Par/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Exp(1))
queue.setService(oclass1, Pareto.fit_mean_and_scv(0.5, 64))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, sink, 1)
model.link(P)
return model
model = gallery_mpar1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: M/Par/1
```

4.2.64 gallery_multitier

A multitier application: presentation, application and database tiers as a closed network.

Processors	3: P0 (ps, $m = 1$), P1 (ps, $m = 1$), P2 (ps, $m = 1$)
Tasks	3: T0 (ref, $m = 1$), T1 (fcfs, $m = 1$), T2 (fcfs, $m = 1$)
Entries	9: E0, E10, E11, E12, E13, E20, E21, E22, E23
Activities	22, host demands A0: 1.0, A1: 1.0, A2: 1.0, A3: 1.0, B0: 1.0, B1: 1.0, B2: 1.0, B3: 1.0, ...
Calls	8 (8 synchronous, 0 asynchronous)

Table 4.65: Features of gallery_multitier.

The example is built and solved as follows.

```
"""
Create a 4-tier J2EE Layered Queueing Network (client/app/database).
Reference LQN with multiple entries per task and or-fork/or-join
activity precedence. Ported from MATLAB gallery_multitier.
```

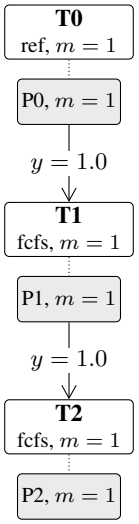

Figure 4.64: Call graph of `gallery_multitier`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
Returns:
    LayeredNetwork: 3-layer (client, application, database) LQN model.
"""
model = LayeredNetwork('testLQN3')
# Layer 1: client
P0 = Processor(model, 'P0', 1, SchedStrategy.PS)
T0 = Task(model, 'T0', 1, SchedStrategy.REF).on(P0)
E0 = Entry(model, 'E0').on(T0)
# Layer 2: application server
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 1, SchedStrategy.FCFS).on(P1)
E10 = Entry(model, 'E10').on(T1)
E11 = Entry(model, 'E11').on(T1)
E12 = Entry(model, 'E12').on(T1)
E13 = Entry(model, 'E13').on(T1)
# Layer 3: database
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T2 = Task(model, 'T2', 1, SchedStrategy.FCFS).on(P2)
E20 = Entry(model, 'E20').on(T2)
E21 = Entry(model, 'E21').on(T2)
E22 = Entry(model, 'E22').on(T2)
E23 = Entry(model, 'E23').on(T2)

    ... (31 source lines omitted; full implementation: python/examples/gallery/)

T1.add_precedence(ActivityPrecedence.OrJoin([B7a, B7b], B8))
T2.add_precedence(ActivityPrecedence.Serial(C0, C1))
T2.add_precedence(ActivityPrecedence.Serial(C3, C4, C5))
return model
model = gallery_multitier()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: LayeredNetwork
```

4.2.65 `gallery_multitier_storage`

The multitier application with an explicit storage tier behind the database.
The example is built and solved as follows.

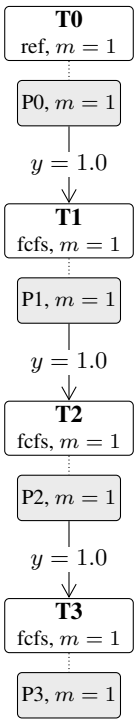

Figure 4.65: Call graph of `gallery_multitier_storage`: 4 tasks over 4 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Processors	4: P0 (ps, $m = 1$), P1 (ps, $m = 1$), P2 (ps, $m = 1$), P3 (ps, $m = 1$)
Tasks	4: T0 (ref, $m = 1$), T1 (fcfs, $m = 1$), T2 (fcfs, $m = 1$), T3 (fcfs, $m = 1$)
Entries	10: E0, E10, E11, E12, E13, E20, E21, E22, E23, E3
Activities	24, host demands A0: 1.0, A1: 1.0, A2: 1.0, A3: 1.0, B0: 1.0, B1: 1.0, B2: 1.0, B3: 1.0, ...
Calls	10 (10 synchronous, 0 asynchronous)

Table 4.66: Features of `gallery_multitier_storage`.

```
"""
Create a 4-tier J2EE LQN with a dedicated cache layer (LRU replacement).
Extends gallery_multitier with a cache layer (CacheTask + ItemEntry) and
cache-access hit/miss precedence. Ported from MATLAB gallery_multitier_storage.
Returns:
    LayeredNetwork: 4-layer LQN model with cache layer.
"""
model = LayeredNetwork('testLQN3_Cache')
# Layer 1: client
P0 = Processor(model, 'P0', 1, SchedStrategy.PS)
T0 = Task(model, 'T0', 1, SchedStrategy.REF).on(P0)
E0 = Entry(model, 'E0').on(T0)
# Layer 2: application server
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 1, SchedStrategy.FCFS).on(P1)
E10 = Entry(model, 'E10').on(T1)
E11 = Entry(model, 'E11').on(T1)
E12 = Entry(model, 'E12').on(T1)
E13 = Entry(model, 'E13').on(T1)
# Layer 3: database
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T2 = Task(model, 'T2', 1, SchedStrategy.FCFS).on(P2)
E20 = Entry(model, 'E20').on(T2)
E21 = Entry(model, 'E21').on(T2)
E22 = Entry(model, 'E22').on(T2)
E23 = Entry(model, 'E23').on(T2)

    ... (41 source lines omitted; full implementation: python/examples/gallery/)

T2.add_precedence(ActivityPrecedence.Serial(C0, C1))
T2.add_precedence(ActivityPrecedence.Serial(C3, C4, C5))
T3.add_precedence(ActivityPrecedence.CacheAccess(D0, [D1a, D1b]))
return model
model = gallery_multitier_storage()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: LayeredNetwork
```

4.2.66 gallery_parm1

The Pareto/M/1 queue: Pareto interarrival times, exponential service, and a single server.

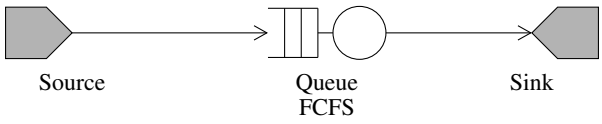

Figure 4.66: Topology of gallery_parm1: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Pareto(2.008,0.5019)
Service	Queue – Class1: Exp(2)

Table 4.67: Features of gallery_parm1.

The example is built and solved as follows.

```
model = Network('Par/M/1')
# Block 1: nodes
```

```
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Pareto.fit_mean_and_scv(1, 64))
queue.setService(oclass1, Exp(2))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, sink, 1)
model.link(P)
return model
model = gallery_parm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Par/M/1
```

4.2.67 `gallery_qn_random`

A randomly generated queueing network, used as a generator-driven parity fixture.

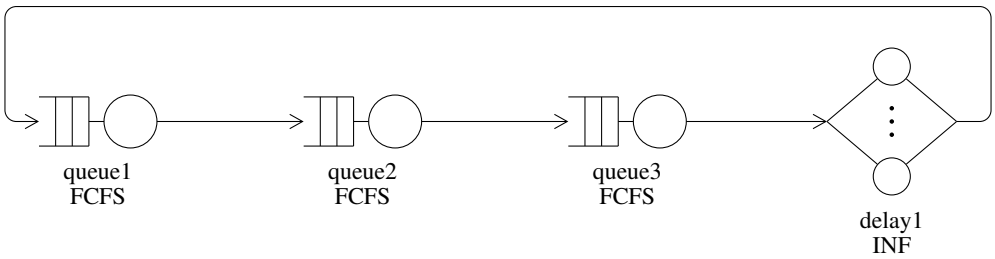

Figure 4.67: Topology of `gallery_qn_random`: a closed network over 4 queueing stations.

Nodes	4: Queue × 3, Delay
Classes	2: CClass1 (closed, $N = 14$), CClass2 (closed, $N = 11$)
Scheduling	queue1: FCFS; queue2: FCFS; queue3: FCFS; delay1: INF
Service	queue1 – CClass1: Exp(1); CClass2: Exp(1) queue2 – CClass1: Exp(1); CClass2: Exp(1) queue3 – CClass1: Exp(1); CClass2: Exp(1) delay1 – CClass1: Exp(1); CClass2: Exp(1)

Table 4.68: Features of `gallery_qn_random`.

The example is built and solved as follows.

```
"""Randomly generated mixed queueing network (reproducible).
Uses NetworkGenerator with a deterministic (cyclic) topology and a fixed
default seed so repeated calls yield the same model. Closed network:
3 queues, 1 delay, 2 closed classes (always stable / solvable).
"""
random.seed(seed)
np.random.seed(seed)
gen = NetworkGenerator(
    sched_strat='fcfs', routing_strat='Probabilities', distribution='Exp',
    cclass_job_load='medium', has_varying_service_rates=False,
    has_multi_server_queues=False, has_random_cs_nodes=False,
    has_multi_chain_cs=False, topology_fcn=cyclic_graph)
return gen.generate(3, 1, 0, 2)
model = gallery_qn_random()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Network
```

4.2.68 gallery_renv_breakdown

A station in a random environment that alternates between an operational and a failed stage.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
"""Random environment: single server with breakdown/repair (UP/DOWN stages).
Returns an Environment whose base model is an M/M/1 queue alternating
between an UP stage (fast service) and a DOWN stage (degraded service).
"""
model = Network('ServerWithFailures')
source = Source(model, 'Arrivals')
queue = Queue(model, 'Server', SchedStrategy.FCFS)
sink = Sink(model, 'Departures')
jobclass = OpenClass(model, 'Jobs')
source.setArrival(jobclass, Exp(0.8))
queue.setService(jobclass, Exp(2.0))
queue.setNumberOfServers(1)
P = model.init_routing_matrix()
P.set(jobclass, jobclass, source, queue, 1.0)
P.set(jobclass, jobclass, queue, sink, 1.0)
model.link(P)
env = Environment('ServerEnv')
env.add_node_failure_repair(model, queue, Exp(0.1), Exp(1.0), Exp(0.5))
env.init()
return env
model = gallery_renv_breakdown()
print('Model built:', type(model).__name__)
```

Running this edition prints

```
Model built: Environment
```

4.2.69 gallery_repairmen

The machine repairman model: machines think, fail, and queue for one repair station.

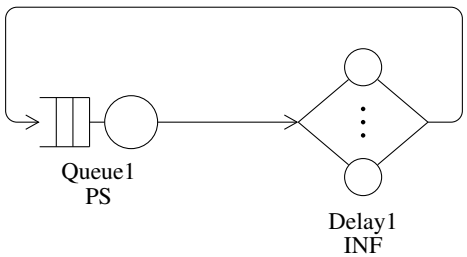

Figure 4.68: Topology of `gallery_repairmen`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 8$)
Scheduling	Queue1: PS; Delay1: INF
Service	Queue1 – Class1: Exp(0.7685) Delay1 – Class1: Exp(0.5)

Table 4.69: Features of `gallery_repairmen`.

The example is built and solved as follows.

```
model = Network('Finite repairmen CQN')
# Block 1: nodes
M = 1
random.seed(seed)
node = []
node.append(Queue(model, 'Queue1', SchedStrategy.PS))
node[0].setNumberOfServers(nservers)
node.append(Delay(model, 'Delay1'))
# Block 2: classes
jobclass = ClosedClass(model, 'Class1', round(random.random() * 10 * M + 3), node[0], 0)
node[0].setService(jobclass, Exp.fit_mean(random.random() + 1))
node[1].setService(jobclass, Exp.fit_mean(2.0))
# Block 3: topology
P = model.init_routing_matrix()
P[jobclass, jobclass] = Network.serial_routing(node)
model.link(P)
return model
model = gallery_repairmen()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Finite repairmen CQN
```

4.2.70 `gallery_replayerm1`

A single queue whose interarrival times are replayed from a measurement trace.

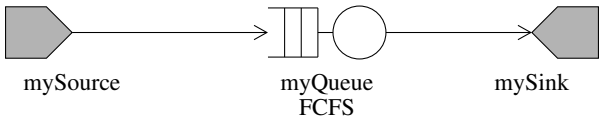

Figure 4.69: Topology of `gallery_replayerm1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: myClass (open)
Scheduling	myQueue: FCFS
Arrivals	mySource – myClass: Replayer(trace)
Service	myQueue – myClass: Exp(29.68)

Table 4.70: Features of `gallery_replayerm1`.

The example is built and solved as follows.

```
if filename is None:
    # Try to find example_trace.txt in common locations
    script_dir = os.path.dirname(os.path.abspath(__file__))
    possible_paths = [
        os.path.join(script_dir, '..', 'gettingstarted', 'example_trace.txt'),
        os.path.join(script_dir, '..', 'basic', 'openQN', 'example_trace.txt'),
    ]
    for path in possible_paths:
        if os.path.exists(path):
            filename = path
            break
    if filename is None:
        raise FileNotFoundError("example_trace.txt not found in expected locations")
model = Network('Trace/M/1')
# Block 1: nodes
source = Source(model, 'mySource')
```

```
queue = Queue(model, 'myQueue', SchedStrategy.FCFS)
sink = Sink(model, 'mySink')
# Block 2: classes
oclass = OpenClass(model, 'myClass')
replayer = Replayer(filename)
source.setArrival(oclass, replayer)
queue.setService(oclass, Exp(3 / replayer.get_mean()))
# Block 3: topology
model.link(Network.serial_routing([source, queue, sink]))
return model
model = gallery_replayerm1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: Trace/M/1
```

4.2.71 `gallery_um1`

The U/M/1 queue: uniform interarrival times, exponential service, and a single server.

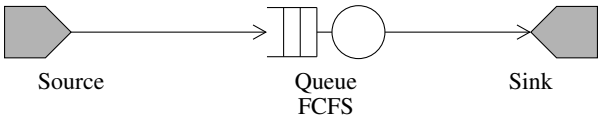

Figure 4.70: Topology of `gallery_um1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Uniform(1,2)
Service	Queue – Class1: Exp(2)

Table 4.71: Features of `gallery_um1`.

The example is built and solved as follows.

```
model = Network('U/M/1')
# Block 1: nodes
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
# Block 2: classes
oclass1 = OpenClass(model, 'Class1')
source.setArrival(oclass1, Uniform(1, 2))
queue.setService(oclass1, Exp(2))
# Block 3: topology
P = model.init_routing_matrix()
P.set(oclass1, oclass1, source, queue, 1)
P.set(oclass1, oclass1, queue, sink, 1)
model.link(P)
return model
model = gallery_um1()
print(f"Model: {model.getName()}")
```

Running this edition prints

```
Model: U/M/1
```

Each of these is also a parity fixture: the four codebases build the same model and the harness compares the metrics solver by solver, so a divergence in any one of them is caught here before it

reaches a larger model.

Chapter 5

Discrete-time models

In a slotted model time advances in slots of fixed length, and every sampled interarrival and service time must fall on the slot lattice. A sample that does not is an error rather than something to round: refusing is what keeps a discrete-time result exact. Within a slot the events are ordered by a fixed phase – completions, internal movement, arrivals, bookkeeping – so that a departure and an arrival in the same slot cannot be confused.

Geometric interarrival and service times give the Geo/Geo/1 queue, whose exact closed form is known, and it is against that form that the solvers are checked. The models live under `python/examples/discrete/`, and slotted mode is requested through the solver options rather than by the model.

```
a = 0.2   # per-slot arrival probability
s = 0.5   # per-slot service completion probability

model = Network('GeoGeo1')
source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')
job_class = OpenClass(model, 'Class1')
source.setArrival(job_class, Geometric(a))
queue.setService(job_class, Geometric(s))
model.link(Network.serialRouting(source, queue, sink))

solver = SolverNC(model)
solver.options.config = {'slotted': True}   # run on the slot lattice
print(solver.getAvgTable())
```

5.1 The slotted models

5.1.1 `dt_bernoulli_loaddep`

A load-dependent Bernoulli server, $p(n) = s\min(n, c)$, the discrete-time multiserver approximation.

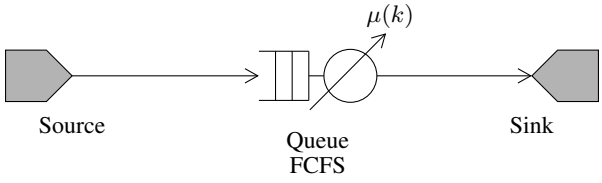

Figure 5.1: Topology of `dt_bernoulli_loaddep`: an open network over 1 queueing station.

The example is built and solved as follows.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Geometric(0.6)
Service	Queue – Class1: Geometric(0.3)
Features	finite buffer 20 at Queue; load-dependent rates at Queue
Solvers	NC

Table 5.1: Features of `dt_bernoulli_loaddep`.

```
a = 0.6    # per-slot arrival probability
s = 0.3    # per-slot service completion probability of one server
c = 3      # servers' worth of capacity
L = 20     # buffer capacity, which bounds the state space

model = Network('LoadDepBernoulli')

source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')

job_class = OpenClass(model, 'Class1')
source.setArrival(job_class, Geometric(a))
queue.setService(job_class, Geometric(s))
queue.setCapacity(L)
alpha = np.minimum(np.arange(1, L + 1), c).astype(float)
queue.setLoadDependence(alpha)          # p(n) = s * min(n,c), example 2.10

model.link(Network.serialRouting(source, queue, sink))

solver = SolverNC(model)
solver.options.config = {'slotted': True}
print(solver.getAvgTable())

r = dqsys_bernoulli1(a, s * alpha, L)
print('time-stationary law pi(0..6)  : %s' % np.round(r['pmf'][:7], 6))
print('arrival law         pi_1(0..6): %s' % np.round(r['arrivalPmf'][:7], 6))
print('no PASTA in discrete time: the two rows above are different laws')
```

Running this edition prints

```
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass    QLen     Util   RespT  ResidT  ArvR  Tput
Source   Class1        0        0       0       0     0   0.6
Queue    Class1   2.0644  0.95402  3.4406  3.4406   0.6   0.6
time-stationary law pi(0..6)  : [0.045977 0.229885 0.402299 0.268199 0.0447   0.00745  0.001242]
arrival law         pi_1(0..6): [1.14943e-01 4.02299e-01 4.02299e-01 6.70500e-02 1.11750e-02 1.86200e-03
 3.10000e-04]
no PASTA in discrete time: the two rows above are different laws
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

5.1.2 `dt_cycle`

A closed cycle of slotted stations.
The example is built and solved as follows.

```
pi(n_1,...,n_J) = prod_j (q_j/p_j)^n_j (1/q_j)^{1{n_j>0}} / G(N,J),
p = [0.5, 0.25, 0.7]   # per-slot service completion probabilities
N = 5                  # jobs cycling

model = Network('BernoulliCycle')
```

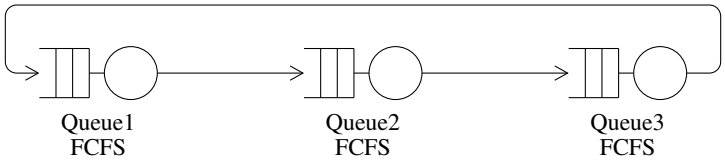

Figure 5.2: Topology of `dt_cycle`: a closed network over 3 queueing stations.

Nodes	3: Queue × 3
Classes	1: Jobs (closed, $N = 5$)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Queue1 – Jobs: Geometric(0.5) Queue2 – Jobs: Geometric(0.25) Queue3 – Jobs: Geometric(0.7)
Solvers	NC

Table 5.2: Features of `dt_cycle`.

```
station = [Queue(model, 'Queue%d' % (j + 1), SchedStrategy.FCFS)
           for j in range(len(p))]

job_class = ClosedClass(model, 'Jobs', N, station[0])
for j in range(len(p)):
    station[j].setService(job_class, Geometric(p[j]))

model.link(Network.serialRouting(*station))

solver = SolverNC(model)
solver.options.config = {'slotted': True}
print(solver.getAvgTable())

# The same quantities from the normalizing constants directly. Throughput
# is G_1(N,J)/G(N,J), the discrete-time counterpart of G(N-1)/G(N), and it
# is the same at every node of the cycle.
lG, G, G1 = dpfqn_nc(p, N)
print('log G(N,J)    = %g' % lG)
print('throughput    = %g jobs per slot' % (G1[N] / G))
print('utilizations = %s' % np.round((G1[N] / G) / np.array(p), 6))
```

Running this edition prints

```
NC analysis [method: dt.cycle; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Jobs      0.72191  0.49656  2.9076  2.9076  0.24828  0.24828
Queue2   Jobs       3.8668  0.99313  15.574  15.574  0.24828  0.24828
Queue3   Jobs      0.41125  0.35469  1.6564  1.6564  0.24828  0.24828
log G(N,J)    = 6.90882
throughput    = 0.248282 jobs per slot
utilizations = [0.496564 0.993128 0.354689]
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

5.1.3 dt_cycle_loaddep

The cycle with a load-dependent slotted server.
The example is built and solved as follows.

```
w_j(n) = prod_{h=1}^{n-1} q_j(h) / prod_{h=1}^{n} p_j(h).
p = [0.5, 0.25, 0.7]
N = 5

model = Network('BernoulliCycleLD')

station = [Queue(model, 'Queue%d' % (j + 1), SchedStrategy.FCFS)
```

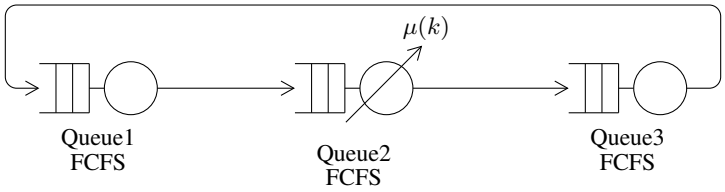

Figure 5.3: Topology of `dt_cycle_loaddep`: a closed network over 3 queueing stations.

Nodes	3: Queue × 3
Classes	1: Jobs (closed, $N = 5$)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Queue1 – Jobs: Geometric(0.5) Queue2 – Jobs: Geometric(0.25) Queue3 – Jobs: Geometric(0.7)
Features	load-dependent rates at Queue2
Solvers	NC

Table 5.3: Features of `dt_cycle_loaddep`.

```
            for j in range(len(p))]

job_class = ClosedClass(model, 'Jobs', N, station[0])
for j in range(len(p)):
    station[j].setService(job_class, Geometric(p[j]))
station[1].setLoadDependence(np.minimum(np.arange(1, N + 1), 2).astype(float))

model.link(Network.serialRouting(*station))

solver = SolverNC(model)
solver.options.config = {'slotted': True}
print(solver.getAvgTable())

# Marginal queue length law of station 2 from the convolution constants.
P = np.tile(np.array(p).reshape(-1, 1), (1, N))
P[1, :] = p[1] * np.minimum(np.arange(1, N + 1), 2)
lG, G, W, Gc, Wa = dpfqn_ncld(P, N)
marg = W[1] * Gc[1][::-1] / G[N]
print('P(X_2 = 0..%d) = %s' % (N, np.round(marg, 6)))
```

Running this edition prints

```
NC analysis [method: dt.cycleld; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Queue1   Jobs       1.7936  0.84515  4.2445  4.2445  0.42258  0.42258
Queue2   Jobs       2.3912  0.94757  5.6586  5.6586  0.42258  0.42258
Queue3   Jobs      0.81523  0.60368  1.9292  1.9292  0.42258  0.42258
P(X_2 = 0..5) = [0.052433 0.204833 0.290108 0.250111 0.156786 0.045729]
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

5.1.4 `dt_cycle_multiclass`

The multiclass cycle.
The example is built and solved as follows.

```
p = [0.5, 0.25, 0.7]
N1 = 3   # jobs in chain 1
N2 = 2   # jobs in chain 2
J = len(p)

model = Network('BernoulliCycleMC')
```

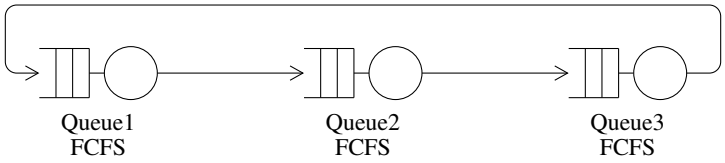

Figure 5.4: Topology of `dt_cycle_multiclass`: a closed network over 3 queueing stations.

Nodes	3: Queue × 3
Classes	2: Chain1 (closed, $N = 3$), Chain2 (closed, $N = 2$)
Scheduling	Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Queue1 – Chain1: Geometric(0.5); Chain2: Geometric(0.5) Queue2 – Chain1: Geometric(0.25); Chain2: Geometric(0.25) Queue3 – Chain1: Geometric(0.7); Chain2: Geometric(0.7)
Solvers	NC

Table 5.4: Features of `dt_cycle_multiclass`.

```
station = [Queue(model, 'Queue%d' % (j + 1), SchedStrategy.FCFS) for j in range(J)]

class1 = ClosedClass(model, 'Chain1', N1, station[0])
class2 = ClosedClass(model, 'Chain2', N2, station[0])
for j in range(J):
    station[j].setService(class1, Geometric(p[j]))
    station[j].setService(class2, Geometric(p[j]))

# Both chains follow the same cycle and never switch class.
cycle = np.roll(np.eye(J), 1, axis=1)
zero = np.zeros((J, J))
P = model.initRoutingMatrix()
P[0][0] = cycle
P[0][1] = zero
P[1][0] = zero
P[1][1] = cycle
model.link(P)

solver = SolverNC(model)
solver.options.config = {'slotted': True}
print(solver.getAvgTable())

lG, G, G1 = dpfqn_nc(p, N1 + N2)
X = G1[N1 + N2] / G
print('aggregate throughput  = %g jobs per slot' % X)
print('per-chain throughput  = %g and %g' % (X * N1 / (N1 + N2), X * N2 / (N1 + N2)))
```

Running this edition prints

```
NC analysis [method: dt.cycle; type: exact, deterministic; lang: python; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT      ArvR      Tput
Queue1   Chain1    0.43315  0.29794  2.9076  2.9076   0.14897   0.14897
Queue1   Chain2    0.28876  0.19863  2.9076  2.9076  0.099313  0.099313
Queue2   Chain1     2.3201  0.59588  15.574  15.574   0.14897   0.14897
Queue2   Chain2     1.5467  0.39725  15.574  15.574  0.099313  0.099313
Queue3   Chain1    0.24675  0.21281  1.6564  1.6564   0.14897   0.14897
Queue3   Chain2     0.1645  0.14188  1.6564  1.6564  0.099313  0.099313
aggregate throughput  = 0.248282 jobs per slot
per-chain throughput  = 0.148969 and 0.0993128
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

5.1.5 `dt_geogeo1`

The Geo/Geo/1 queue: Bernoulli arrivals and Bernoulli service on the slot lattice, against the closed form of Daduna.

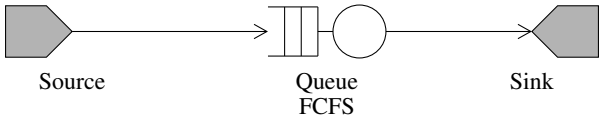

Figure 5.5: Topology of `dt_geogeo1`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Geometric(0.2)
Service	Queue – Class1: Geometric(0.5)
Solvers	NC

Table 5.5: Features of `dt_geogeo1`.

The example is built and solved as follows.

```
a = 0.2   # per-slot arrival probability
s = 0.5   # per-slot service completion probability

model = Network('GeoGeo1')

source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')

job_class = OpenClass(model, 'Class1')
source.setArrival(job_class, Geometric(a))
queue.setService(job_class, Geometric(s))

model.link(Network.serialRouting(source, queue, sink))

solver = SolverNC(model)
solver.options.config = {'slotted': True}   # run on the slot lattice
print(solver.getAvgTable())

print('closed form: E[N] = %g, E[T] = %g slots'
      % (a * (1 - a) / (s - a), (1 - a) / (s - a)))

# The same numbers straight from the single-queue formula.
r = dqsys_bernoulli1(a, s)
print('E[N] = %g, U = %g, X = %g per slot'
      % (r['meanQueueLength'], r['utilization'], r['throughput']))
```

Running this edition prints

```
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass     QLen  Util   RespT  ResidT  ArvR  Tput
Source   Class1          0     0       0       0     0   0.2
Queue    Class1    0.53333   0.4  2.6667  2.6667   0.2   0.2
closed form: E[N] = 0.533333, E[T] = 2.66667 slots
E[N] = 0.533333, U = 0.4, X = 0.2 per slot
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

5.1.6 `dt_geogeo1_loss`

The Geo/Geo/1/L loss system, and the loss probability.
The example is built and solved as follows.

```
a = 0.2   # per-slot arrival probability
```

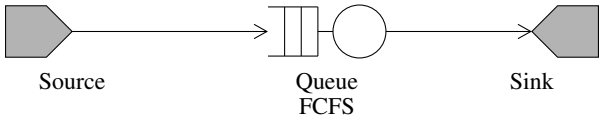

Figure 5.6: Topology of `dt_geogeo1_loss`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Class1 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Geometric(0.2)
Service	Queue – Class1: Geometric(0.5)
Features	finite buffer 4 at Queue
Solvers	NC

Table 5.6: Features of `dt_geogeo1_loss`.

```
s = 0.5    # per-slot service completion probability
L = 4      # buffer capacity in jobs

model = Network('GeoGeo1L')

source = Source(model, 'Source')
queue = Queue(model, 'Queue', SchedStrategy.FCFS)
sink = Sink(model, 'Sink')

job_class = OpenClass(model, 'Class1')
source.setArrival(job_class, Geometric(a))
queue.setService(job_class, Geometric(s))
queue.setCapacity(L)

model.link(Network.serialRouting(source, queue, sink))

solver = SolverNC(model)
solver.options.config = {'slotted': True}
print(solver.getAvgTable())

r = dqsys_bernoulli1(a, s, L)
print('queue length law   : %s' % np.round(r['pmf'], 6))
print('loss probability   : %g' % r['lossProb'])
print('carried throughput : %g of the %g offered per slot' % (r['throughput'], a))
```

Running this edition prints

```
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: python; env: python] completed in <t
    >s.
Station  JobClass     QLen    Util    RespT  ResidT  ArvR     Tput
Source   Class1          0       0        0       0     0      0.2
Queue    Class1    0.52256  0.3985   2.6226  2.6226   0.2  0.19925
queue length law   : [0.601504 0.300752 0.075188 0.018797 0.003759]
loss probability   : 0.0037594
carried throughput : 0.199248 of the 0.2 offered per slot
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `NC`.

Superposing two slotted streams produces batches, not a slotted stream of the same kind, so a tandem of slotted queues is the first model in which a discrete-time implementation can go wrong while the single queue still looks right. `dt_cycle` exists for that reason.

Chapter 6

Solver-specific examples

Most examples in this manual are written around a model and let the solver vary. The scripts of this chapter do the opposite: they are written around one solver, and print what it builds rather than only what it concludes. They live under `python/examples/solvers/`.

The CTMC solver is the natural subject, because its intermediate objects are the model: the state space, the infinitesimal generator and the stationary distribution are all printable, and the averages are read off them. Printing them is how one checks that a model means what it was intended to mean.

6.0.1 `ctmc_spn_mm1`

The same M/M/1 system written as a queueing model and as a Petri net, and the two generators compared.

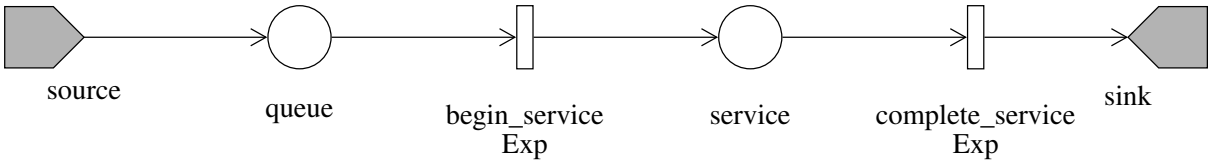

Figure 6.1: Topology of `ctmc_spn_mm1`: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

Nodes	6: Source, Place × 2, Transition × 2, Sink
Classes	1: jobs (open)
Arrivals	source – jobs: Exp(0.8)
Features	2 transitions with 1 firing mode each
Solvers	CTMC

Table 6.1: Features of `ctmc_spn_mm1`.

The example is built and solved as follows.

```
"""Build and solve M/M/1 as SPN with CTMC."""
print("=" * 70)
print("M/M/1 Queue as Stochastic Petri Net (SPN)")
print("=" * 70)
# Create network
model = Network('spn_mm1')
# SPN Components
# Source for open arrivals
source = Source(model, 'source')
```

```
sink = Sink(model, 'sink')
# SPN places represent queue states
p_queue = Place(model, 'queue')      # Customers waiting
p_service = Place(model, 'service')  # Customer in service
# SPN transitions represent events
t_begin = Transition(model, 'begin_service')    # Start service
t_finish = Transition(model, 'complete_service') # Finish service
# Job class
jobclass = OpenClass(model, 'jobs')
# Define arrival process (lambda=0.8, mean=1.25)
source.setArrival(jobclass, Exp.fit_mean(1/0.8))
# Configure "begin_service" Transition
# This transition models a customer starting service
# - Requires: 1 customer in queue, 0 in service
# - Effect: move from queue->service
mode_begin = t_begin.add_mode('begin')
t_begin.set_distribution(mode_begin, Exp.fit_mean(1.0))  # Rate=1.0

    ... (72 source lines omitted; full implementation: python/examples/solvers/)

print("  Differences from theory are due to CTMC state space truncation")
print("  (open networks with infinite arrivals truncated at cutoff=10)")
print("=" * 70)
return model, solver
GlobalConstants.set_verbose(VerboseLevel.STD)
model, solver = spn_mm1_example()
```

Running this edition prints

```
======================================================================
M/M/1 Queue as Stochastic Petri Net (SPN)
======================================================================

Solving with CTMC solver...
(Note: open network with CTMC uses state space truncation)
CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  Stochcomp: 198 immediate states eliminated, 66 non-immediate states remain.
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.

CTMC Results for SPN M/M/1:
Station  JobClass    QLen    Util   RespT  ResidT     ArvR     Tput
source   jobs          0       0       0       0        0  0.74788

... (19 captured-output lines omitted; rerun the source example for the full output)

======================================================================
Summary
======================================================================
? CTMC successfully analyzes M/M/1 modeled as SPN
? SPN feature support in CTMC is working correctly
  Differences from theory are due to CTMC state space truncation
  (open networks with infinite arrivals truncated at cutoff=10)
======================================================================
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

6.0.2 `ctmc_spn_vs_nc_closed_qn`

A closed network solved exactly by CTMC over its Petri net form and by the normalizing constant method.
The example is built and solved as follows.

```
                    Network, Place, Queue, SchedStrategy, Transition,
                    VerboseLevel)
"""The scalar `column` of the row naming `station`, or None."""
```

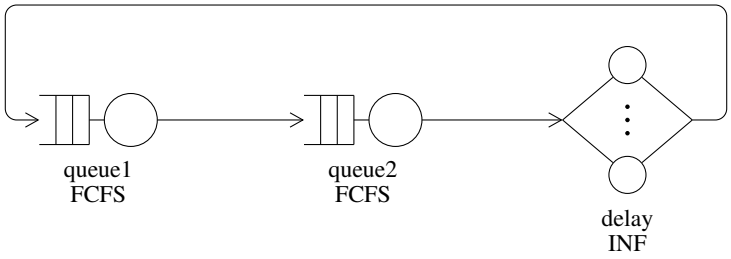

Figure 6.2: Topology of `ctmc_spn_vs_nc_closed_qn`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	1: jobs (closed, $N = 3$)
Scheduling	queue1: FCFS; queue2: FCFS; delay: INF
Service	queue1 – jobs: Exp(1) queue2 – jobs: Exp(0.8) delay – jobs: Exp(0.5)

Table 6.2: Features of `ctmc_spn_vs_nc_closed_qn`.

```
rows = table[table['Station'].astype(str) == station]
if len(rows) == 0:
    return None
return float(np.asarray(rows[column], dtype=float).sum())
GlobalConstants.set_verbose(VerboseLevel.SILENT)

print(BAR)
print('Closed QN: CTMC (SPN) vs NC Solver Comparison')
print(BAR)

print('\n' + BAR)
print('Approach 1: Traditional Closed QN (Solved with NC Solver)')
print(BAR)

model_qn = Network('closed_qn_traditional')

queue1 = Queue(model_qn, 'queue1', SchedStrategy.FCFS)
queue2 = Queue(model_qn, 'queue2', SchedStrategy.FCFS)
delay = Delay(model_qn, 'delay')

jobclass = ClosedClass(model_qn, 'jobs', 3, queue1)    # 3 jobs, start at queue1

queue1.setService(jobclass, Exp.fitMean(1.0))          # mu1 = 1.0

    ... (120 source lines omitted; full implementation: python/examples/solvers/)

print('  - Total population conserved (=3 jobs)')
print('  - System throughput matches')
print('  - Individual station metrics match')
print('\nOK CTMC successfully models closed QN as SPN')
print('OK Different modeling approaches yield same answers')
print(BAR)
```

Running this edition prints

```
======================================================================
Closed QN: CTMC (SPN) vs NC Solver Comparison
======================================================================

======================================================================
Approach 1: Traditional Closed QN (Solved with NC Solver)
======================================================================

Solving with NC (Normalizing Constant) solver...
NC Results:
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
queue1   jobs      0.80954  0.53644  1.5091  1.5091  0.53644  0.53644
queue2   jobs       1.1176  0.67055  2.0833  2.0833  0.53644  0.53644
delay    jobs       1.0729   1.0729       2       2  0.53644  0.53644
```

```
======================================================================

... (49 captured-output lines omitted; rerun the source example for the full output)

OK Product-form property verified:
  - Total population conserved (=3 jobs)
  - System throughput matches
  - Individual station metrics match

OK CTMC successfully models closed QN as SPN
OK Different modeling approaches yield same answers
======================================================================
```

6.0.3 `ctmc_tandem_mm1`

The generator of a two-station tandem, its state space and its stationary distribution, printed in full.

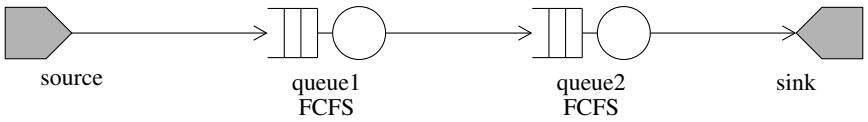

Figure 6.3: Topology of `ctmc_tandem_mm1`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: jobs (open)
Scheduling	queue1: FCFS; queue2: FCFS
Arrivals	source – jobs: Exp(0.6)
Service	queue1 – jobs: Exp(1) queue2 – jobs: Exp(1.2)
Solvers	CTMC

Table 6.3: Features of `ctmc_tandem_mm1`.

The example is built and solved as follows.

```python
"""Build and solve tandem M/M/1 queues with CTMC."""
print("=" * 70)
print("Tandem M/M/1 Queues (Series of Two Stations)")
print("=" * 70)
# Create network
model = Network('tandem_mm1')
# Define nodes
# Source sends jobs to Queue1
source = Source(model, 'source')
# Queue 1: lambda? = 0.6, mu? = 1.0 (rho? = 0.6)
queue1 = Queue(model, 'queue1', SchedStrategy.FCFS)
# Queue 2: lambda? = 0.6, mu? = 1.2 (rho? = 0.5)
queue2 = Queue(model, 'queue2', SchedStrategy.FCFS)
# Sink receives departures
sink = Sink(model, 'sink')
# Job class
jobclass = OpenClass(model, 'jobs')
# Define arrival and service processes
# External arrivals: lambda = 0.6 (mean = 1/0.6 ~= 1.667)
source.setArrival(jobclass, Exp.fit_mean(1/0.6))
# Queue 1 service: mu? = 1.0 (mean = 1.0)
queue1.setService(jobclass, Exp.fit_mean(1.0))
# Queue 2 service: mu? = 1.2 (mean = 1/1.2 ~= 0.833)
queue2.setService(jobclass, Exp.fit_mean(1/1.2))
# Define routing: Source -> Queue1 -> Queue2 -> Sink
R = model.init_routing_matrix()

    ... (107 source lines omitted; full implementation: python/examples/solvers/)
```

```
print("? Results match theoretical predictions")
print("? CTMC feature support is working correctly!")
print("=" * 70)
return model, solver
GlobalConstants.set_verbose(VerboseLevel.STD)
model, solver = tandem_mm1_example()
```

Running this edition prints

```
======================================================================
Tandem M/M/1 Queues (Series of Two Stations)
======================================================================

Solving tandem M/M/1 system with CTMC...
Parameters:
  Station 1: lambda=0.6, mu=1.0, rho=0.6
  Station 2: lambda=0.6, mu=1.2, rho=0.5

CTMC solver using state space cutoff = 10 for open/mixed model.
python/line_solver/api/solvers/ctmc/handler.py:2298: UserWarning: State space truncation may cause
     inaccurate results. Consider varying cutoff to assess sensitivity.
  warnings.warn(
  CTMC has 1 SCCs; absorption seeded from initial state
CTMC analysis [method: default/sync; type: exact, deterministic; lang: python; env: python] completed in <
     t>s.
CTMC Results for Tandem M/M/1:
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput

... (58 captured-output lines omitted; rerun the source example for the full output)


? CTMC successfully analyzes tandem M/M/1-M/M/1 system
? Product-form property verified:
  - Each station behaves independently as M/M/1
  - System decomposition holds
? Results match theoretical predictions
? CTMC feature support is working correctly!
======================================================================
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `CTMC`.

Two of the three scripts build the same system twice, once as a queueing model and once as a stochastic Petri net, and compare the generators. That is the strongest available check on either notation: the two descriptions are independent, and if they agree state by state then neither has been misunderstood.

Chapter 7

Model interchange

A model can be written to, and read back from, a JSON document. The format is the interchange between the codebases and the wire protocol of the LDES engine, which is driven entirely by it: the client serializes `model.json`, the engine solves it and returns `result.json`.

The examples under `python/examples/json/` are round-trip fixtures: each builds a model that uses one group of wire keys, writes it, reads it back and checks that the reconstructed model has the same structure. What a round trip must preserve is the declared state of the model; what it must not carry is the derived state, which is recomputed on load.

```
model = json_hetero_servers()       # a model using one group of wire keys
save_model(model, 'model.json')
reloaded = load_model('model.json')
```

7.1 The fixtures

7.1.1 `json_balking_retrial`

Balking thresholds and retrial orbits over the wire.

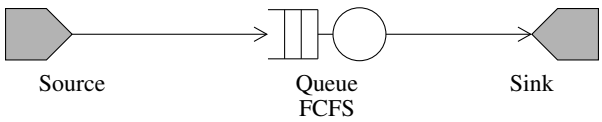

Figure 7.1: Topology of `json_balking_retrial`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Exp(0.5)
Service	Queue – Class1: Exp(2); Class2: Exp(3)
Features	finite buffer 15 at Queue; drop rule {'Class1': 'retrialWithLimit', 'Class2': 'retrialWithLimit'} at Queue

Table 7.1: Features of `json_balking_retrial`.

The example is built and solved as follows.

```
def json_balking_retrial():
    """Open 2-class network with balking, retrial, and patience."""
    model = Network('Balking_Retrial')
    source = Source(model, 'Source')
```

```
    queue = Queue(model, 'Queue', SchedStrategy.FCFS)
    sink = Sink(model, 'Sink')
    queue.set_capacity(15)
    class1 = OpenClass(model, 'Class1')
    class2 = OpenClass(model, 'Class2')
    source.set_arrival(class1, Exp(1))
    source.set_arrival(class2, Exp(0.5))
    queue.set_service(class1, Exp(2))
    queue.set_service(class2, Exp(3))
    # Class1: balking based on queue length
    queue.set_balking(class1, BalkingStrategy.QUEUE_LENGTH,
                      [(5, 10, 0.3), (11, float('inf'), 1.0)])
    # Class1: patience (reneging)
    queue.set_patience(class1, Exp(0.1))
    # Class2: retrial with max attempts
    queue.set_retrial(class2, Exp(0.5), 3)
    model.link(Network.serial_routing(source, queue, sink))
    return model
```

The captured run completed successfully and produced no console output.

7.1.2 `json_classcap_droprule`

Per-class capacities and drop rules.

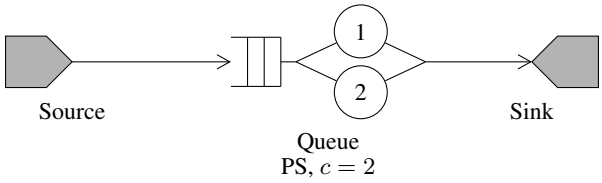

Figure 7.2: Topology of `json_classcap_droprule`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: PS ($c = 2$)
Arrivals	Source – Class1: Exp(1); Class2: Exp(0.5)
Service	Queue – Class1: Exp(3); Class2: Exp(2)
Features	finite buffer 20 at Queue; load-dependent rates at Queue; drop rule {'Class1': 'drop', 'Class2': 'blockingAfterService'} at Queue

Table 7.2: Features of `json_classcap_droprule`.

The example is built and solved as follows.

```
def json_classcap_droprule():
    """Open 2-class network with per-class capacity, drop rules, and load-dependence."""
    model = Network('ClassCap_DropRule')
    source = Source(model, 'Source')
    queue = Queue(model, 'Queue', SchedStrategy.PS)
    sink = Sink(model, 'Sink')
    queue.set_number_of_servers(2)
    queue.set_capacity(20)
    class1 = OpenClass(model, 'Class1')
    class2 = OpenClass(model, 'Class2')
    source.set_arrival(class1, Exp(1))
    source.set_arrival(class2, Exp(0.5))
    queue.set_service(class1, Exp(3))
    queue.set_service(class2, Exp(2))
    # Per-class capacity
    queue.set_class_capacity(class1, 8)
    queue.set_class_capacity(class2, 15)
```

```
    # Per-class drop rules. WAITQ is not usable here: no solver honours "wait
    # upstream" for an open class at a plain finite capacity (CTMC drops the
    # arrival, JMT blocks at the source and ignores the cap), so the capacity
    # refresh rejects that combination. BAS is the honoured blocking policy.
    queue.set_drop_rule(class1, DropStrategy.DROP)
    queue.set_drop_rule(class2, DropStrategy.BAS)
    # Load-dependent scaling
    queue.set_load_dependence(np.array([1.0, 0.9, 0.8, 0.7]))
    model.link(Network.serial_routing(source, queue, sink))
    return model
```

The captured run completed successfully and produced no console output.

7.1.3 `json_fcr_details`

The finite capacity region block, with its members, limits and drop rules.

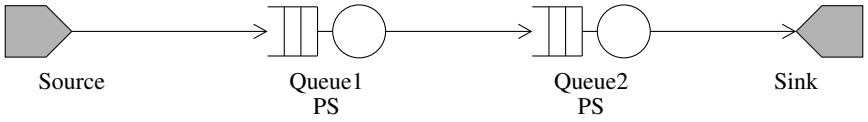

Figure 7.3: Topology of `json_fcr_details`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: PS; Queue2: PS
Arrivals	Source – Class1: Exp(1); Class2: Exp(0.5)
Service	Queue1 – Class1: Exp(3); Class2: Exp(2) Queue2 – Class1: Exp(4); Class2: Exp(3)

Table 7.3: Features of `json_fcr_details`.

The example is built and solved as follows.

```
def json_fcr_details():
    """Open 2-class network with a Finite Capacity Region."""
    model = Network('FCR_Details')
    source = Source(model, 'Source')
    queue1 = Queue(model, 'Queue1', SchedStrategy.PS)
    queue2 = Queue(model, 'Queue2', SchedStrategy.PS)
    sink = Sink(model, 'Sink')
    class1 = OpenClass(model, 'Class1')
    class2 = OpenClass(model, 'Class2')
    source.set_arrival(class1, Exp(1))
    source.set_arrival(class2, Exp(0.5))
    queue1.set_service(class1, Exp(3))
    queue1.set_service(class2, Exp(2))
    queue2.set_service(class1, Exp(4))
    queue2.set_service(class2, Exp(3))
    model.link(Network.serial_routing(source, queue1, queue2, sink))
    # Finite Capacity Region
    fcr = Region([queue1, queue2], [class1, class2])
    fcr.set_global_max_jobs(15)
    fcr.set_class_max_jobs(class1, 8)
    fcr.set_class_max_jobs(class2, 10)
    fcr.set_class_weight(class1, 1.0)
    fcr.set_class_weight(class2, 2.0)
    fcr.set_class_size(class1, 1)
    fcr.set_class_size(class2, 3)
    return model
```

The captured run completed successfully and produced no console output.

7.1.4 json_hetero_servers

Heterogeneous server types at one station, their per-type service rates and the policy that orders them.

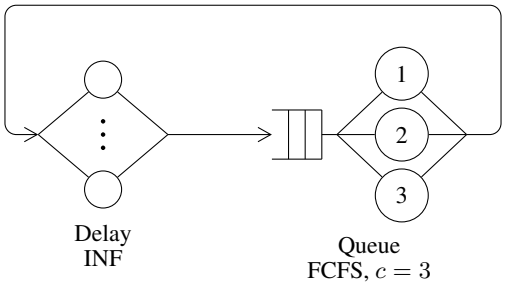

Figure 7.4: Topology of `json_hetero_servers`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 3$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue: FCFS ($c = 3$)
Service	Delay – Class1: Exp(1); Class2: Exp(2) Queue – Class1: Exp(3); Class2: Exp(2)

Table 7.4: Features of `json_hetero_servers`.

The example is built and solved as follows.

```
def json_hetero_servers():
    """Closed 2-class network with heterogeneous servers."""
    model = Network('HeteroServers')
    delay = Delay(model, 'Delay')
    queue = Queue(model, 'Queue', SchedStrategy.FCFS)
    queue.set_number_of_servers(3)
    class1 = ClosedClass(model, 'Class1', 3, delay)
    class2 = ClosedClass(model, 'Class2', 2, delay)
    delay.set_service(class1, Exp(1))
    delay.set_service(class2, Exp(2))
    # Default service (required before hetero setup)
    queue.set_service(class1, Exp(3))
    queue.set_service(class2, Exp(2))
    # Heterogeneous server types
    fast = ServerType('Fast', 2, [class1, class2])
    slow = ServerType('Slow', 1, [class1])
    queue.add_server_type(fast)
    queue.add_server_type(slow)
    queue.set_hetero_sched_policy(HeteroSchedPolicy.ORDER)
    queue.set_hetero_service(class1, fast, Exp(5))
    queue.set_hetero_service(class2, fast, Exp(3))
    queue.set_hetero_service(class1, slow, Exp(1))
    P = model.init_routing_matrix()
    P.set(class1, class1, delay, queue, 1.0)
    P.set(class1, class1, queue, delay, 1.0)
    P.set(class2, class2, delay, queue, 1.0)
    P.set(class2, class2, queue, delay, 1.0)
    model.link(P)
    return model
```

The captured run completed successfully and produced no console output.

7.1.5 json_join_deadline

Join deadlines and the quorum they imply.
The example is built and solved as follows.

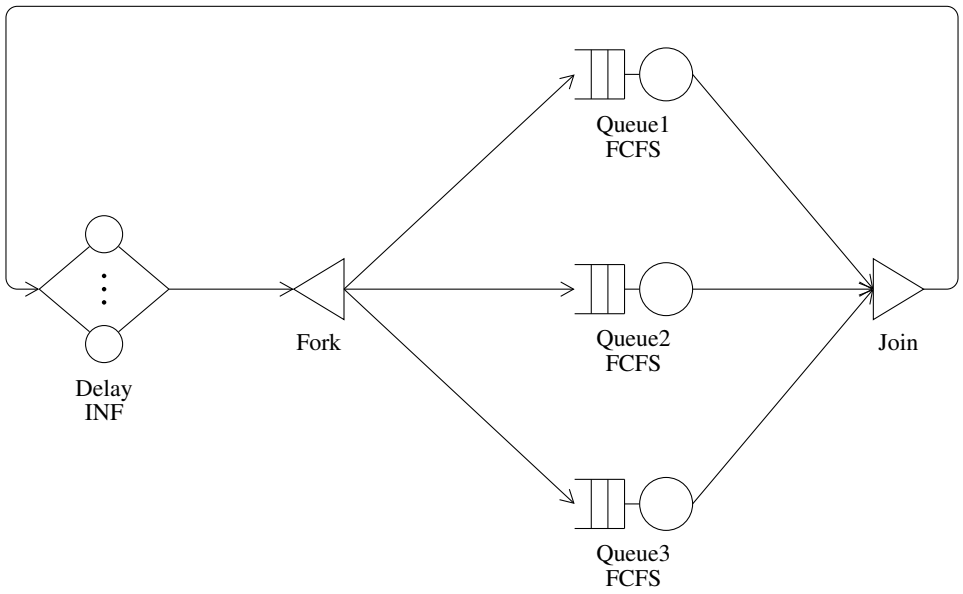

Figure 7.5: Topology of `json_join_deadline`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue $\times$ 3, Delay, Fork, Join
Classes	1: Class1 (closed, $N = 5$)
Scheduling	Delay: INF; Queue1: FCFS; Queue2: FCFS; Queue3: FCFS
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(3) Queue2 – Class1: Exp(4) Queue3 – Class1: Exp(2)

Table 7.5: Features of `json_join_deadline`.

```
def json_join_deadline():
    """Closed 1-class fork-join with quorum join and deadline."""
    model = Network('JoinDeadline')
    delay = Delay(model, 'Delay')
    fork = Fork(model, 'Fork')
    queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
    queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
    queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS)
    join = Join(model, 'Join', fork)
    class1 = ClosedClass(model, 'Class1', 5, delay)
    class1.setDeadline(5.0)
    delay.set_service(class1, Exp(1))
    queue1.set_service(class1, Exp(3))
    queue2.set_service(class1, Exp(4))
    queue3.set_service(class1, Exp(2))
    join.set_strategy(class1, JoinStrategy.QUORUM)
    join.set_required(class1, 2)
    P = model.init_routing_matrix()
    P.set(class1, class1, delay, fork, 1.0)
    P.set(class1, class1, fork, queue1, 1.0)
    P.set(class1, class1, fork, queue2, 1.0)
    P.set(class1, class1, fork, queue3, 1.0)
    P.set(class1, class1, queue1, join, 1.0)
    P.set(class1, class1, queue2, join, 1.0)
    P.set(class1, class1, queue3, join, 1.0)
    P.set(class1, class1, join, delay, 1.0)
    model.link(P)
    return model
```

The captured run completed successfully and produced no console output.

7.1.6 `json_signal_classes`

G-network signal classes.

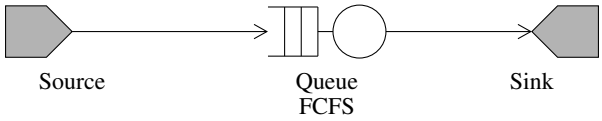

Figure 7.6: Topology of `json_signal_classes`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Signal1 (signal)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(2); Signal1: Exp(0.5)
Service	Queue – Class1: Exp(5); Signal1: Immediate

Table 7.6: Features of `json_signal_classes`.

The example is built and solved as follows.

```
def json_signal_classes():
    """Open network with G-network negative signal."""
    model = Network('SignalClasses')
    source = Source(model, 'Source')
    queue = Queue(model, 'Queue', SchedStrategy.FCFS)
    sink = Sink(model, 'Sink')
    class1 = OpenClass(model, 'Class1')
    signal1 = OpenSignal(model, 'Signal1', SignalType.NEGATIVE)
    signal1.forJobClass(class1)
    signal1.setRemovalPolicy(RemovalPolicy.RANDOM)
    source.set_arrival(class1, Exp(2))
```

```
    source.set_arrival(signal1, Exp(0.5))
    queue.set_service(class1, Exp(5))
    queue.set_service(signal1, Immediate())
    P = model.init_routing_matrix()
    P.set(class1, class1, source, queue, 1.0)
    P.set(class1, class1, queue, sink, 1.0)
    P.set(signal1, signal1, source, queue, 1.0)
    P.set(signal1, signal1, queue, sink, 1.0)
    model.link(P)
    return model
```

The captured run completed successfully and produced no console output.

The fixtures are grouped by wire key rather than by model family on purpose: a key that no fixture exercises is a key that no round trip tests, and grouping them this way makes the gap visible.

Chapter 8

Inference and estimation

Inference runs the modeling process backwards: given measurements of a system – arrival rates, utilizations, response times, or a raw trace – it estimates the model parameters that would produce them. The unknown parameters are marked in the model by a distribution with a `NaN` parameter, and the estimator fills them in.

The estimators divide into three families: likelihood-based (MLE and its quasi-likelihood variant), recursive (the extended Kalman filter and its smoother, which track a parameter that drifts), and Bayesian (sampling and variational, which return a posterior rather than a point estimate). The changepoint scripts test all three against a workload that changes abruptly, which is where a recursive estimator earns its cost. The reference material is in the Inference chapter of the LINE user manual.

The scripts live under `python/examples/inference/`, and each one generates a synthetic dataset from known parameters, so the estimate can be compared against the truth.

```
# The unknown demands are declared as NaN and estimated from the data
node[1].setService(jobclass1, Exp(float('nan')))    # true demand 0.1
node[1].setService(jobclass2, Exp(float('nan')))    # true demand 0.3

# Synthetic dataset consistent with the model
n = 100
util_samples = 0.25 + np.random.randn(n) * 0.02
respt1_samples = 0.1 / (1 - util_samples)
respt2_samples = 0.3 / (1 - util_samples)
```

8.1 The estimators

8.1.1 `est_ekf_changepoint`

The filter under a changepoint in the workload.

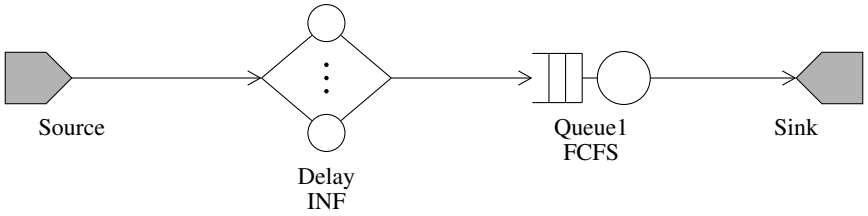

Figure 8.1: Topology of `est_ekf_changepoint`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
model = Network('model')
```

Nodes	4: Source, Queue, Delay, Sink
Classes	1: Class1 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(2.903)

Table 8.1: Features of `est_ekf_changepoint`.

```
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None]
jobclass[0] = OpenClass(model, 'Class1', 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(1.0))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
model.link(P)
n = 200; changeT = 100; D1 = 0.3; D2 = 0.6; lam = 1.0; noise = 0.02
ts = np.arange(1, n + 1, dtype=float)
arvr_samples = lam * np.ones(n) + np.random.randn(n) * noise
util_samples = np.zeros(n)
respt_samples = np.zeros(n)
for t in range(n):
    D = D1 if t < changeT else D2
    U = lam * D + np.random.randn() * noise
    U = max(0.01, min(0.95, U))
    util_samples[t] = U

    ... (24 source lines omitted; full implementation: python/examples/inference/)

print(f'{"t":>6}  {"Estimated":>10}  {"True":>10}')
for cp in checkpoints:
    if W <= cp <= n:
        est = estimates[cp - W]
        trueD = D1 if cp <= changeT else D2
        print(f'{cp:6d}  {est:10.4f}  {trueD:10.4f}')
```

Running this edition prints

```
=== EKF Change-Point Detection (warm-start) ===
True demand: D=0.3 (t<=100), D=0.6 (t>100)
Window size: 30

     t   Estimated        True
    30      0.3445      0.3000
    50      0.3445      0.3000
    80      0.3445      0.3000
   100      0.3445      0.3000
   110      0.3445      0.6000
   120      0.3445      0.6000
   130      0.3445      0.6000
   150      0.3445      0.6000
   180      0.3445      0.6000
   200      0.3445      0.6000
```

8.1.2 `est_ekf_closed`

The closed-network filter.
The example is built and solved as follows.

```
np.random.seed(1)
```

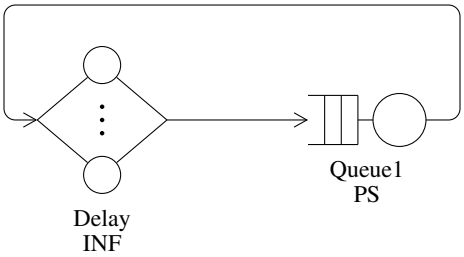

Figure 8.2: Topology of `est_ekf_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(2.5)
Solvers	MVA

Table 8.2: Features of `est_ekf_closed`.

```
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None]
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
model.link(P)
n = 100
ts = np.arange(1, n + 1, dtype=float)
arvr_samples = 1.5 * np.ones(n) - np.random.rand(n) * 0.1
util_samples = 0.4 * arvr_samples
respt_samples = 0.4 / (1 - util_samples)
lambda1 = SampledMetric(MetricType.ArvR, ts, arvr_samples, node[1], jobclass[0])
respT1 = SampledMetric(MetricType.RespT, ts, respt_samples, node[1], jobclass[0])
util = SampledMetric(MetricType.Util, ts, util_samples, node[1])
# EKF
print('\n=== EKF Estimator ===')
options = ParamEstimator.defaultOptions()
options['method'] = 'ekf'
se = ParamEstimator(model, options)

    ... (20 source lines omitted; full implementation: python/examples/inference/)

estVal_auto = se2.estimateAt(node[1])
print(f'Auto demand: Class1={estVal_auto[0]:.4f} (true=0.4000)')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
=== EKF Estimator ===
EKF demand: Class1=0.6630 (true=0.4000)

=== autoMethod selection ===
autoMethod selected: ubo
Required metrics for ubo: ArvR (per-class) + RespT (per-class) + Util (aggregate)
Auto demand: Class1=0.4000 (true=0.4000)

SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
```

```
    t>s.
Station  JobClass     QLen    Util    RespT   ResidT    ArvR    Tput
Delay    Class1     1.3208  1.3208       1        1  1.3208  1.3208
Queue1   Class1    0.67925  0.5283  0.51429  0.51429  1.3208  1.3208
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.3 `est_ekf_open`

An extended Kalman filter tracking the demands of an open network online.

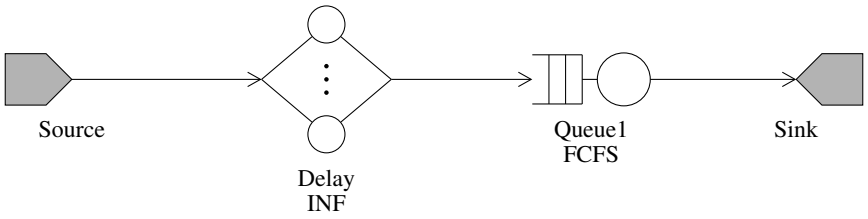

Figure 8.3: Topology of `est_ekf_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	1: Class1 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(2.093)
Solvers	MVA

Table 8.3: Features of `est_ekf_open`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None]
jobclass[0] = OpenClass(model, 'Class1', 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(1.0))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
model.link(P)
n = 100
ts = np.arange(1, n + 1, dtype=float)
arvr_samples = 1.0 * np.ones(n) + np.random.randn(n) * 0.05
util_samples = 0.4 * np.ones(n) + np.random.randn(n) * 0.02
util_samples = np.clip(util_samples, 0.01, 0.95)
respt_samples = 0.4 / (1 - util_samples)
lambda1 = SampledMetric(MetricType.ArvR, ts, arvr_samples, node[1], jobclass[0])
respT1 = SampledMetric(MetricType.RespT, ts, respt_samples, node[1], jobclass[0])
util = SampledMetric(MetricType.Util, ts, util_samples, node[1])
print('\n=== EKF Estimator (Open Network) ===')

    ... (7 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
```

```
print(f'EKF demand: Class1={estVal[0]:.4f} (true=0.4000)')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
=== EKF Estimator (Open Network) ===
EKF demand: Class1=0.4778 (true=0.4000)

SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass    QLen     Util  RespT  ResidT  ArvR  Tput
Delay    Class1        1        1      1       1     1     1
Queue1   Class1    0.915  0.47781  0.915   0.915     1     1
Source   Class1        0        0      0       0     0     1
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.4 `est_erps_closed`

The closed-network smoother.

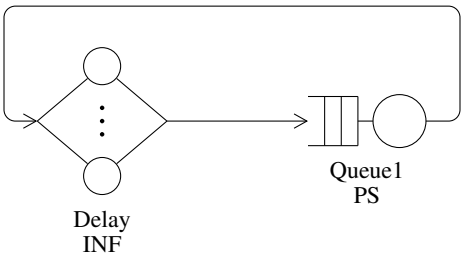

Figure 8.4: Topology of `est_erps_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 3$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(10); Class2: Exp(3.333)
Solvers	MVA

Table 8.4: Features of `est_erps_closed`.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 3, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
```

```
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvr1_samples = np.ones(n) - np.random.rand(n) * 0.15
arvr2_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15
util_samples = 0.1 * arvr1_samples + 0.3 * arvr2_samples
respt1_samples = 0.1 / (1 - util_samples)
respt2_samples = 0.3 / (1 - util_samples)
aqlen1_samples = 1 + util_samples / (1 - util_samples)

    ... (15 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.1 0.3]
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.83698   0.83698        1        1  0.83698  0.83698
Delay    Class2     1.9821    1.9821        1        1   1.9821   1.9821
Queue1   Class1    0.16302  0.083698  0.19477  0.19477  0.83698  0.83698
Queue1   Class2     1.0179   0.59463  0.51353  0.51353   1.9821   1.9821
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.5 `est_erps_open`

The Rauch-Tung-Striebel smoother on an open network.

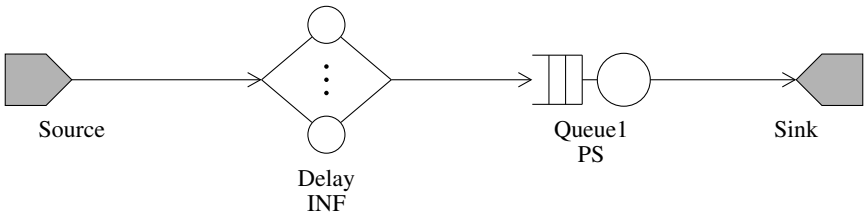

Figure 8.5: Topology of `est_erps_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF; Queue1: PS
Arrivals	Source – Class1: Exp(0.1); Class2: Exp(0.05)
Service	Delay – Class1: HyperExp(0.5,0.5;3,10); Class2: HyperExp(0.5,0.5;2,8) Queue1 – Class1: Exp(10); Class2: Exp(3.333)
Solvers	MVA

Table 8.5: Features of `est_erps_open`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
```

```
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None] * 2
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
node[0].setService(jobclass[0], HyperExp(0.5, 3.0, 10.0))
node[0].setService(jobclass[1], HyperExp(0.5, 2.0, 8.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(0.1))
node[2].setArrival(jobclass[1], Exp(0.05))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[3], 1.0)
P.set(jobclass[1], jobclass[1], node[2], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvrl_samples = np.ones(n) - np.random.rand(n) * 0.15

    ... (20 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.1 0.3]

SOLVER: MVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
Station  JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.021667  0.021667  0.21667  0.21667   0.1   0.1
Delay    Class2    0.015625  0.015625   0.3125   0.3125  0.05  0.05
Queue1   Class1    0.010157      0.01  0.10157  0.10157   0.1   0.1
Queue1   Class2    0.015235     0.015  0.30471  0.30471  0.05  0.05
Source   Class1           0         0        0        0     0   0.1
Source   Class2           0         0        0        0     0  0.05
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.6 `est_mcmc_closed`

Markov chain Monte Carlo sampling of the posterior over the demands.

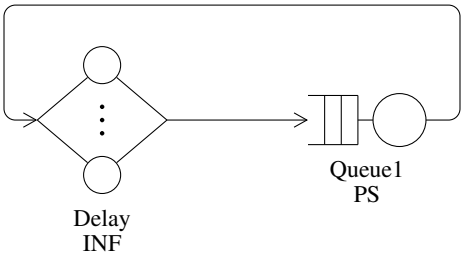

Figure 8.6: Topology of `est_mcmc_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 3$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(0.5703); Class2: Exp(0.5703)
Solvers	MVA

Table 8.6: Features of `est_mcmc_closed`.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 3, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 500
ts = np.arange(1, n + 1, dtype=float)
qlen_samples = 1.5 + np.random.rand(n) * 0.5
ql = SampledMetric(MetricType.QLen, ts, qlen_samples, node[1])
print('\n=== MCMC Estimator ===')
options = ParamEstimator.defaultOptions()
options['method'] = 'mcmc'
se = ParamEstimator(model, options)
se.addSamples(ql)
se.interpolate()
estVal = se.estimateAt(node[1])
print(f'MCMC demands: Class1={estVal[0]:.4f}, Class2={estVal[1]:.4f}')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
=== MCMC Estimator ===
MCMC demands: Class1=1.7535, Class2=1.7535

SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.22805  0.22805       1       1  0.22805  0.22805
Delay    Class2    0.34208  0.34208       1       1  0.34208  0.34208
Queue1   Class1      1.772  0.39989  7.7701  7.7701  0.22805  0.22805
Queue1   Class2     2.6579  0.59983  7.7698  7.7698  0.34208  0.34208
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.7 `est_mle_open`

Maximum likelihood estimation of the service demands of an open network from arrival rate, utilization and response time samples.

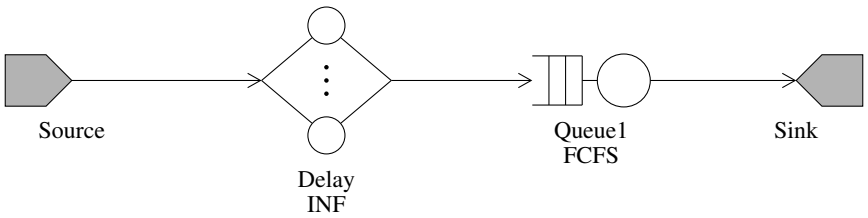

Figure 8.7: Topology of `est_mle_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Exp(0.5)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(13.34); Class2: Exp(9.113)
Solvers	MVA

Table 8.7: Features of `est_mle_open`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None] * 2
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(1.0))
node[2].setArrival(jobclass[1], Exp(0.5))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[3], 1.0)
P.set(jobclass[1], jobclass[1], node[2], node[0], 1.0)
model.link(P)
n = 100
ts = np.arange(1, n + 1, dtype=float)
arvr1_samples = 1.0 * np.ones(n) + np.random.randn(n) * 0.05

    ... (20 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print(f'MLE demands: Class1={estVal[0]:.4f} (true=0.1000), Class2={estVal[1]:.4f} (true
    =0.3000)')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
=== MLE Estimator (Open Network) ===
MLE demands: Class1=0.0750 (true=0.1000), Class2=0.1097 (true=0.3000)

SOLVER: MVA
MVA analysis [method: default/qd; type: approximate, deterministic; lang: python; env: python] completed
    in <t>s.
```

```
Station  JobClass      QLen      Util      RespT    ResidT  ArvR  Tput
Delay    Class1          1         1          1         1     1     1
Delay    Class2        0.5       0.5          1         1   0.5   0.5
Queue1   Class1   0.088339  0.074963   0.088339  0.088339     1     1
Queue1   Class2   0.061553  0.054865    0.12311   0.12311   0.5   0.5
Source   Class1          0         0          0         0     0     1
Source   Class2          0         0          0         0     0   0.5
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.8 `est_qmle_closed`

Quasi-likelihood estimation on a closed network.

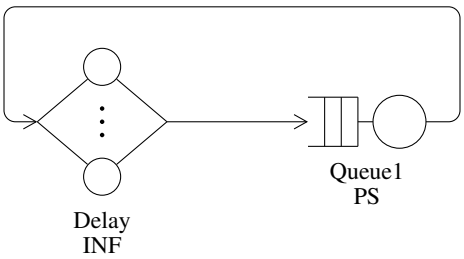

Figure 8.8: Topology of `est_qmle_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 2$), Class2 (closed, $N = 3$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(9.553); Class2: Exp(4.837)
Solvers	MVA

Table 8.8: Features of `est_qmle_closed`.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 2, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 3, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
qlen1_samples = 0.3 * np.ones(n) + np.random.rand(n) * 0.1
qlen2_samples = 0.8 * np.ones(n) + np.random.rand(n) * 0.1
options = ParamEstimator.defaultOptions()
options['method'] = 'qmle'
se = ParamEstimator(model, options)
ql1 = SampledMetric(MetricType.QLen, ts, qlen1_samples, node[1], jobclass[0])
```

```
    ... (4 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.10467796 0.20674601]
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1    1.6716   1.6716        1        1  1.6716  1.6716
Delay    Class2    2.2072   2.2072        1        1  2.2072  2.2072
Queue1   Class1   0.32836  0.17498  0.19643  0.19643  1.6716  1.6716
Queue1   Class2   0.79283  0.45632   0.3592   0.3592  2.2072  2.2072
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.9 `est_trace_closed`

Estimation from a raw arrival and completion trace rather than from aggregate samples.
No topology is drawn for this example: the captured run yielded no `Network` or `LayeredNetwork` to draw one from, either because the script builds a model of another kind (an environment or a workflow) or because it builds it inside a helper it does not expose.
The example is built and solved as follows.

```
np.random.seed(1)
N = 5
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None]
jobclass[0] = ClosedClass(model, 'Class1', N, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
model.link(P)
# Generate synthetic trace data
n = 200
arrival_times = np.sort(np.random.rand(n) * 100)
response_times = 0.5 + np.random.rand(n) * 0.3
tput_samples = np.ones(n) * (N / (1.0 + 0.5))
arvData = SampledMetric(MetricType.ArvR, arrival_times, arrival_times, node[1], jobclass[0])
arvData.setTrace()
rtData = SampledMetric(MetricType.RespT, arrival_times, response_times, node[1], jobclass[0])
rtData.setTrace()
tputData = SampledMetric(MetricType.Tput, arrival_times, tput_samples, node[1], jobclass[0])
# MLPS
print('\n=== MLPS Estimator ===')

    ... (49 source lines omitted; full implementation: python/examples/inference/)

print(f'FMLPS: {fmlps_val:.4f}')
print(f'Gibbs: {gibbs_val:.4f}')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

The run exceeded the capture time budget. No output is shown, and this edition makes no completion claim.

8.1.10 `est_ubo_changepoint`

The variational estimator under a changepoint.

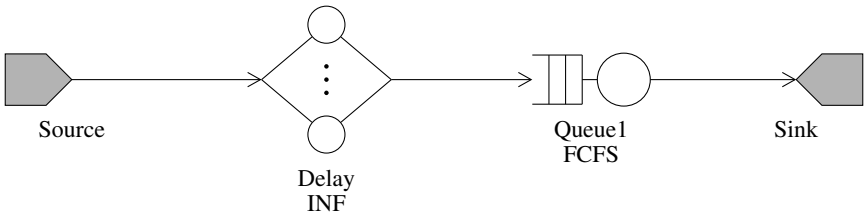

Figure 8.9: Topology of `est_ubo_changepoint`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Exp(0.5)
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(3.338); Class2: Exp(5.018)

Table 8.9: Features of `est_ubo_changepoint`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None] * 2
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(1.0))
node[2].setArrival(jobclass[1], Exp(0.5))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[3], 1.0)
P.set(jobclass[1], jobclass[1], node[2], node[0], 1.0)
model.link(P)
n = 200; changeT = 100
D1_before = 0.1; D1_after = 0.3; D2_val = 0.2
lambda1 = 1.0; lambda2 = 0.5; noise = 0.01

    ... (37 source lines omitted; full implementation: python/examples/inference/)

checkpoints = [W, 50, 80, 100, 110, 120, 130, 150, 180, 200]
print(f'{"t":>6}  {"Est D1":>8}  {"True D1":>8}  {"Est D2":>8}  {"True D2":>8}')
for cp in checkpoints:
    if W <= cp <= n:
        trueD1 = D1_before if cp <= changeT else D1_after
        print(f'{cp:6d}  {est1[cp-W]:8.4f}  {trueD1:8.4f}  {est2[cp-W]:8.4f}  {D2_val:8.4f}')
```

Running this edition prints

```
=== UBO Change-Point Detection (2-class) ===
Class 1: D=0.1 (t<=100), D=0.3 (t>100)
Class 2: D=0.2 (constant)
Window size: 30

     t    Est D1   True D1    Est D2   True D2
    30    0.0996    0.1000    0.1992    0.2000
    50    0.1005    0.1000    0.2006    0.2000
    80    0.0991    0.1000    0.1993    0.2000
   100    0.1006    0.1000    0.2006    0.2000
   110    0.1852    0.3000    0.1906    0.2000
   120    0.2483    0.3000    0.1924    0.2000
   130    0.3002    0.3000    0.2003    0.2000
   150    0.3003    0.3000    0.2002    0.2000
   180    0.3006    0.3000    0.2005    0.2000
   200    0.2995    0.3000    0.1993    0.2000
```

8.1.11 `est_ubo_closed`

The closed-network version.

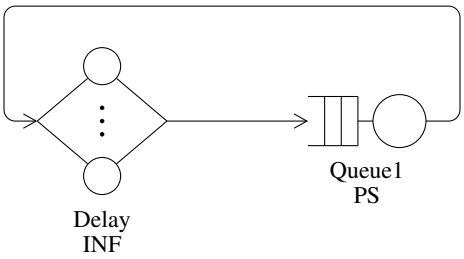

Figure 8.10: Topology of `est_ubo_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(10); Class2: Exp(3.333)
Solvers	MVA

Table 8.10: Features of `est_ubo_closed`.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 30
ts = np.arange(1, n + 1, dtype=float)
```

```
arvr2_samples = np.ones(n) - np.random.rand(n) * 0.15
arvr1_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15
util_samples = 0.1 * arvr1_samples + 0.3 * arvr2_samples
respt1_samples = 0.1 / (1 - util_samples)
respt2_samples = 0.3 / (1 - util_samples)
options = ParamEstimator.defaultOptions()

    ... (13 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.1 0.3]
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1     0.8666   0.8666        1        1  0.8666  0.8666
Delay    Class2     1.4216   1.4216        1        1  1.4216  1.4216
Queue1   Class1     0.1334  0.08666  0.15393  0.15393  0.8666  0.8666
Queue1   Class2    0.57838  0.42648  0.40685  0.40685  1.4216  1.4216
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.12 `est_ubo_open`

Variational Bayesian estimation on an open network.

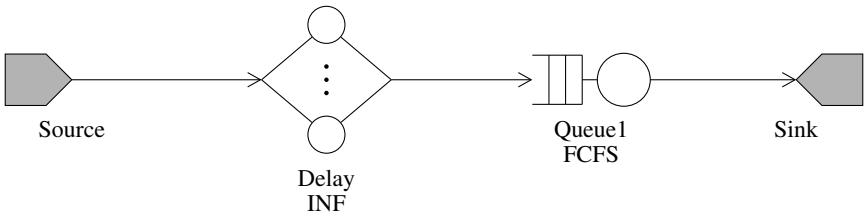

Figure 8.11: Topology of `est_ubo_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(0.1); Class2: Exp(0.05)
Service	Delay – Class1: HyperExp(0.5,0.5;3,10); Class2: HyperExp(0.5,0.5;2,8) Queue1 – Class1: Exp(10); Class2: Exp(3.333)
Solvers	MVA

Table 8.11: Features of `est_ubo_open`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
```

```
jobclass = [None] * 2
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
node[0].setService(jobclass[0], HyperExp(0.5, 3.0, 10.0))
node[0].setService(jobclass[1], HyperExp(0.5, 2.0, 8.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(0.1))
node[2].setArrival(jobclass[1], Exp(0.05))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[3], 1.0)
P.set(jobclass[1], jobclass[1], node[2], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvr1_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15

    ... (18 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.1 0.3]
SOLVER: MVA
MVA analysis [method: default/qd; type: approximate, deterministic; lang: python; env: python] completed
     in <t>s.
Station  JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.021667  0.021667  0.21667  0.21667   0.1   0.1
Delay    Class2    0.015625  0.015625   0.3125   0.3125  0.05  0.05
Queue1   Class1    0.010564      0.01  0.10564  0.10564   0.1   0.1
Queue1   Class2    0.015282     0.015  0.30564  0.30564  0.05  0.05
Source   Class1           0         0        0        0     0   0.1
Source   Class2           0         0        0        0     0  0.05
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.13 `est_ubo_variants_closed`

The variants of the variational estimator, side by side.

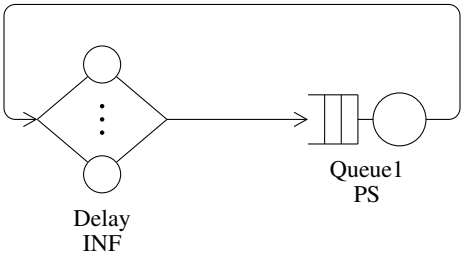

Figure 8.12: Topology of `est_ubo_variants_closed`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
```

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 3$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(11.21); Class2: Exp(4.547)
Solvers	MVA

Table 8.12: Features of `est_ubo_variants_closed`.

```
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 3, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvr1_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15
arvr2_samples = np.ones(n) - np.random.rand(n) * 0.15
util_samples = 0.1 * arvr1_samples + 0.3 * arvr2_samples
respt1_samples = 0.1 / (1 - util_samples)
respt2_samples = 0.3 / (1 - util_samples)
lambda1 = SampledMetric(MetricType.ArvR, ts, arvr1_samples, node[1], jobclass[0])

    ... (30 source lines omitted; full implementation: python/examples/inference/)

print(f'UBO:  Class1={estVal_ubo[0]:.4f}, Class2={estVal_ubo[1]:.4f}')
print(f'MLE:  Class1={estVal_mle[0]:.4f}, Class2={estVal_mle[1]:.4f}')
solver = MVA(model)
print(f'\nSOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
=== UBO Estimator ===
UBO demands: Class1=0.1000, Class2=0.3000

=== MLE Estimator ===
MLE demands: Class1=0.0892, Class2=0.2199

=== Comparison ===
True demands: Class1=0.1000, Class2=0.3000
UBO:  Class1=0.1000, Class2=0.3000
MLE:  Class1=0.0892, Class2=0.2199

SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.86763   0.86763        1        1  0.86763  0.86763
Delay    Class2     2.2361    2.2361        1        1   2.2361   2.2361
Queue1   Class1    0.13237  0.077365  0.15257  0.15257  0.86763  0.86763
Queue1   Class2     0.7639   0.49182  0.34162  0.34162   2.2361   2.2361
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.14 `est_ubr_changepoint`

The regression estimator under a changepoint.

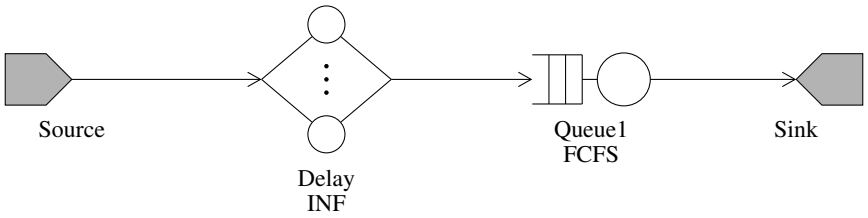

Figure 8.13: Topology of `est_ubr_changepoint`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	1: Class1 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(1)
Service	Delay – Class1: Exp(1) Queue1 – Class1: Exp(1.682)

Table 8.13: Features of `est_ubr_changepoint`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None]
jobclass[0] = OpenClass(model, 'Class1', 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(1.0))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
model.link(P)
n = 200
changeT = 100
D1 = 0.3
D2 = 0.6
lam = 1.0
noise = 0.02
ts = np.arange(1, n + 1, dtype=float)
arvr_samples = lam * np.ones(n) + np.random.randn(n) * noise
util_samples = np.zeros(n)
respt_samples = np.zeros(n)

    ... (25 source lines omitted; full implementation: python/examples/inference/)

print(f'{"------":>6}  {"----------":>10}  {"----------":>10}')
for cp in checkpoints:
    if W <= cp <= n:
        est = estimates[cp - W]
        trueD = D1 if cp <= changeT else D2
        print(f'{cp:6d}  {est:10.4f}  {trueD:10.4f}')
```

Running this edition prints

```
=== UBR Change-Point Detection ===
True demand: D=0.3 (t<=100), D=0.6 (t>100)
Window size: 30
```

```
     t   Estimated        True
------  ----------  ----------
    30      0.2985      0.3000
    50      0.2998      0.3000
    80      0.2997      0.3000
   100      0.3068      0.3000
   110      0.4062      0.6000
   120      0.5029      0.6000
   130      0.5959      0.6000
   150      0.6003      0.6000
   180      0.5895      0.6000
   200      0.5945      0.6000
```

8.1.15 `est_ubr_closed`

The closed-network version.

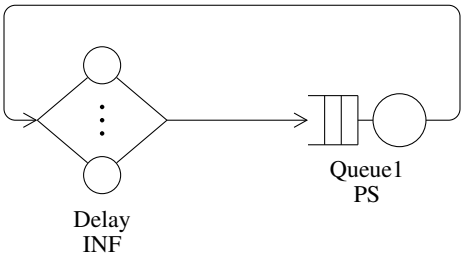

Figure 8.14: Topology of `est_ubr_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	2: Class1 (closed, $N = 1$), Class2 (closed, $N = 2$)
Scheduling	Delay: INF; Queue1: PS
Service	Delay – Class1: Exp(1); Class2: Exp(1) Queue1 – Class1: Exp(2.502); Class2: Exp(13.93)
Solvers	MVA

Table 8.14: Features of `est_ubr_closed`.

The example is built and solved as follows.

```
np.random.seed(1)
model = Network('model')
node = [None] * 2
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.PS)
jobclass = [None] * 2
jobclass[0] = ClosedClass(model, 'Class1', 1, node[0], 0)
jobclass[1] = ClosedClass(model, 'Class2', 2, node[0], 0)
node[0].setService(jobclass[0], Exp.fitMean(1.0))
node[0].setService(jobclass[1], Exp.fitMean(1.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[1].setService(jobclass[1], Exp(float('nan')))
P = model.initRoutingMatrix()
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[0], 1.0)
P.set(jobclass[1], jobclass[1], node[0], node[1], 1.0)
P.set(jobclass[1], jobclass[1], node[1], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvr1_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15
arvr2_samples = 3 * np.ones(n) - np.random.rand(n) * 0.25
util_samples = np.ones(n) - np.random.rand(n) * 0.05
util1_samples = 0.4 * (2 * np.ones(n) - np.random.rand(n) * 0.15)
options = ParamEstimator.defaultOptions()
```

```
options['method'] = 'ubr'

    ... (10 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.39966    0.07179307]
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
Delay    Class1    0.68656  0.68656         1         1  0.68656  0.68656
Delay    Class2     1.8193   1.8193         1         1   1.8193   1.8193
Queue1   Class1    0.31344  0.27439   0.45653   0.45653  0.68656  0.68656
Queue1   Class2     0.1807  0.13061  0.099322  0.099322   1.8193   1.8193
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.16 `est_ubr_open`

The regression-based estimator on an open network.

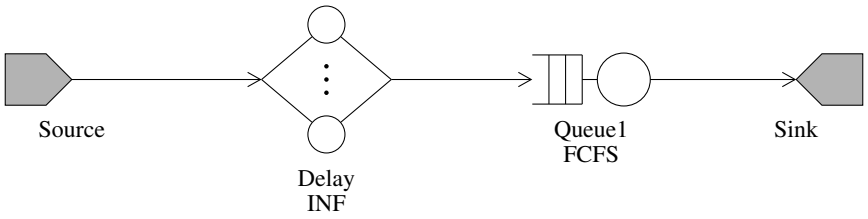

Figure 8.15: Topology of `est_ubr_open`: an open network over 2 queueing stations.

Nodes	4: Source, Queue, Delay, Sink
Classes	1: Class1 (open)
Scheduling	Delay: INF; Queue1: FCFS
Arrivals	Source – Class1: Exp(0.1)
Service	Delay – Class1: HyperExp(0.5,0.5;3,10) Queue1 – Class1: Exp(1.973)
Solvers	MVA

Table 8.15: Features of `est_ubr_open`.

The example is built and solved as follows.

```
model = Network('model')
node = [None] * 4
node[0] = Delay(model, 'Delay')
node[1] = Queue(model, 'Queue1', SchedStrategy.FCFS)
node[2] = Source(model, 'Source')
node[3] = Sink(model, 'Sink')
jobclass = [None]
jobclass[0] = OpenClass(model, 'Class1', 0)
node[0].setService(jobclass[0], HyperExp(0.5, 3.0, 10.0))
node[1].setService(jobclass[0], Exp(float('nan')))
node[2].setArrival(jobclass[0], Exp(0.1))
P = model.initRoutingMatrix()
```

```
P.set(jobclass[0], jobclass[0], node[0], node[1], 1.0)
P.set(jobclass[0], jobclass[0], node[1], node[3], 1.0)
P.set(jobclass[0], jobclass[0], node[2], node[0], 1.0)
model.link(P)
n = 1000
ts = np.arange(1, n + 1, dtype=float)
arvrate_samples = 2 * np.ones(n) - np.random.rand(n) * 0.15
util_samples = np.ones(n) - np.random.rand(n) * 0.05
respt_samples = 0.7 / (1 - util_samples)
estoptions = ParamEstimator.defaultOptions()
estoptions['method'] = 'ubr'
se = ParamEstimator(model, estoptions)
lambda1 = SampledMetric(MetricType.ArvR, ts, arvrate_samples, node[1], jobclass[0])
respT1 = SampledMetric(MetricType.RespT, ts, respt_samples, node[1], jobclass[0])

    ... (5 source lines omitted; full implementation: python/examples/inference/)

estVal = se.estimateAt(node[1])
print('Estimated demands:', estVal)
solver = MVA(model)
print(f'SOLVER: {solver.getName()}')
avgTable = solver.getAvgTable()
print(avgTable)
```

Running this edition prints

```
Estimated demands: [0.50679508]
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass      QLen      Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.021667  0.021667  0.21667  0.21667   0.1   0.1
Queue1   Class1    0.053385   0.05068  0.53385  0.53385   0.1   0.1
Source   Class1           0         0        0        0     0   0.1
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

8.1.17 `est_vi_closed`

Variational inference over the transition counts of a closed loop, from noisy queue-length readings, returning a Gamma posterior per rate rather than a point estimate.

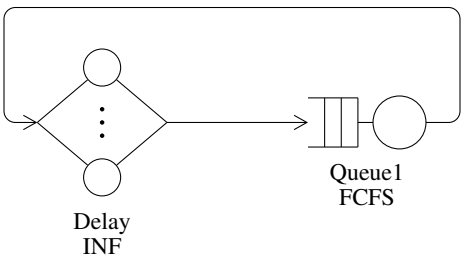

Figure 8.16: Topology of `est_vi_closed`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Class1 (closed, $N = 10$)
Scheduling	Delay: INF; Queue1: FCFS
Service	Delay – Class1: Exp(0.5) Queue1 – Class1: Exp(1.994)
Solvers	MVA

Table 8.16: Features of `est_vi_closed`.

The example is built and solved as follows.

```
Method 'vi' (I. Perez, G. Casale, Adv. Appl. Prob. 53(3), 2021) infers service
rates from NOISY QUEUE-LENGTH READINGS taken over time: each reading is exact
with probability 1-epsilon and uniform over the remaining feasible values
otherwise. Unlike the other estimators it returns a conjugate Gamma POSTERIOR
per rate, not only a point estimate.
It reads the queue lengths of EVERY station, not only the estimated one: the
transition counts the method is written in are pinned by the whole picture.
                        MetricType, SolverMVA)
N = 10
# define model, with the queue rate to be estimated
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
cl = ClosedClass(model, 'Class1', N, delay, 0)
delay.setService(cl, Exp(0.5))
queue.setService(cl, Exp(2.0))     # starting point of the estimate
model.link(Network.serialRouting(delay, queue))
# queue-length readings, one per unit time, 10% of them faulty
ts = np.arange(1, 11, dtype=float)
qlen = np.array([1, 2, 3, 2, 4, 3, 5, 4, 3, 4], dtype=float)
# estimate the service rate of the queue
se = ParamEstimator(model)
se.options['method'] = 'vi'
se.options['epsilon'] = 0.1       # probability that a reading is faulty
se.options['prior_shape'] = 2.0   # Gamma prior shape; the rate is set from the model
se.options['ngrid'] = 51          # time grid of the backward and forward passes

    ... (8 source lines omitted; full implementation: python/examples/inference/)

print('posterior service rate ~ Gamma(%.4f, %.4f), mean %.4f'
      % (post[0, 0], post[0, 1], post[0, 0] / post[0, 1]))
print('evidence lower bound over the iterations:',
      np.round(np.asarray(se.options['bound']), 3))
# solve the model the estimate has been written into
print(SolverMVA(model).getAvgTable())
```

Running this edition prints

```
infer_variational: iteration 1, lower bound -89.392514, max rate 60.000
infer_variational: iteration 2, lower bound -164.046307, max rate 60.000
infer_variational: iteration 3, lower bound -163.710848, max rate 60.000
infer_variational: iteration 4, lower bound -174.650086, max rate 60.000
infer_variational: iteration 5, lower bound -163.233639, max rate 60.000
Estimated demand: 0.50152363
posterior service rate ~ Gamma(20.8739, 10.4688), mean 1.9939
evidence lower bound over the iterations: [ -89.393 -164.046 -163.711 -174.65  -163.234]
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Station  JobClass    QLen     Util    RespT  ResidT    ArvR    Tput
Delay    Class1    3.9671   3.9671        2       2  1.9835  1.9835
Queue1   Class1    6.0329  0.99479   3.0415  3.0415  1.9835  1.9835
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

Chapter 9

Optimization

Besides evaluating a given model, LINE can solve design problems on queueing networks. While the solvers answer evaluation questions (given a model, what are its performance metrics?), the optimization layer answers design questions: how many servers, what service rates, which routing probabilities, or what job populations optimize a cost or performance objective subject to service-level constraints. The reference material on decision variables, objectives, constraints and solvers is in the Optimization chapter of the LINE user manual; this chapter collects the worked examples.

The relevant scripts are included under the `examples/opt/` folder and each runs in seconds. The general structure of an optimization script consists of four blocks:

1. Definition of the LINE network model
2. Declaration of the decision variables
3. Specification of the objective and constraints
4. Solution

Every public method is offered both in CamelCase and snake_case, following LINE's dual naming convention (e.g., `addVariable` and `add_variable` are equivalent).

9.1 Sizing an M/M/c queue

The M/M/c queue is a classic model of a service center with c parallel servers fed by a common infinite-capacity buffer. In this example, we wish to find the minimum number of servers such that the utilization does not exceed 50%. Jobs arrive at rate $\lambda = 3$ job/s and each server works at rate $\mu = 1$ job/s, hence the utilization with c servers is $\rho = \lambda/(c\mu) = 3/c$. It is known from theory that the optimum is therefore $c^* = 6$; we wish to recover this value. The following script (`opt_server_sizing.py`) solves the problem with the default differential evolution solver, at a cost of 10 units per server:

```
from line_solver import Network, Queue, Source, Sink, OpenClass, Exp, SchedStrategy
from line_solver import (OptimizationProblem, ServerAllocation,
                         MinimizeCost, UtilizationConstraint)

## Block 1: LINE model (Source -> Queue -> Sink)
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))    # lambda = 3
queue.setService(jobs, Exp(1.0))     # mu = 1 per server
model.addLink(source, queue)
model.addLink(queue, sink)
```

```
## Block 2: decision variables
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))

## Block 3: objective and constraints
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))

## Block 4: solution
result = problem.solve(seed=42)
```

Obtaining the following output:

```
Optimal servers : 6
Cost            : 60.0
Feasible        : True
LINE evaluations: 10
```

Note that only 10 distinct configurations required a LINE solve: the evaluator caches results at the level of the decoded configuration, so the hundreds of candidate vectors explored by differential evolution collapse onto the 10 possible server counts.

9.2 Exact sizing by bisection

The sizing problem of Tutorial 1 has a special structure: feasibility is monotone in the decision variable, since adding servers can only decrease utilization. In such cases `BisectionSolver` finds the exact optimum in $O(\log n)$ LINE evaluations. Model and problem definition are identical to Tutorial 1 (`opt_bisection_sizing.py`); only the solution block changes:

```
from line_solver import BisectionSolver

result = BisectionSolver(problem).solve()
```

Obtaining the following output:

```
Optimal servers : 6
Cost            : 60.0
LINE evaluations: 4 (vs hundreds for DE)
```

9.3 Continuous service rate tuning

We now choose the cheapest service rate, a continuous decision, meeting a response time SLA on an M/M/1 queue. With $\lambda = 3$, the response time is $R = 1/(\mu - 3)$, so $R \leq 0.5$ requires $\mu \geq 5$; the rate is billed at 20 cost units per unit, hence the optimum is $\mu^* = 5$ at cost 100. In the script (`opt_service_rate.py`), the model is an M/M/1 queue built as in Tutorial 1 with initial rate $\mu = 4$, and the problem is declared as:

```
problem = OptimizationProblem(model)
problem.add_variable(ServiceRate(queue, jobs, bounds=(3.5, 8.0)))
problem.set_objective(MinimizeCost(rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))

result = problem.solve(seed=42, max_iterations=60)
```

Obtaining the following output:

```
Optimal rate    : 5.001 (theory: 5.0)
Cost            : 100.0
Feasible        : True
```

9.4 Load balancing across heterogeneous servers

An arrival stream of rate $\lambda = 2$ must be split between a fast queue ($\mu_1 = 4$) and a slow queue ($\mu_2 = 2.5$) so that the mean end-to-end response time is minimized. Modeling each queue as M/M/1, the mean number of jobs in the system is $N(p) = \frac{\rho_1}{1-\rho_1} + \frac{\rho_2}{1-\rho_2}$ with $\rho_1 = p\lambda/\mu_1$ and $\rho_2 = (1-p)\lambda/\mu_2$, and by Little's law the system response time is $N(p)/\lambda$. Setting $dN/dp = 0$ yields $p^* \approx 0.743$ with response time 0.4264: the optimal policy does not saturate the fast server. The script (`opt_load_balancing.py`) declares the routing split as the decision variable:

```
problem = OptimizationProblem(model)
problem.add_variable(RoutingProbabilities(jobs, source=source,
                                          targets=[fast, slow]))
problem.set_objective(MinimizeSystemResponseTime(jobs))

result = problem.solve(seed=42, max_iterations=60)
probs = result.getVariableValue('Jobs_routing_from_S')
```

Obtaining the following output:

```
Fraction to Fast: 0.736 (theory: 0.743)
Fraction to Slow: 0.264
System RT       : 0.4248 (theory: 0.4264)
```

The system response time is computed by Little's law from the chain queue lengths and throughput, which remains correct on parallel topologies.

9.5 Population sizing in a closed network

An interactive system is modeled as a closed network with a Delay station (mean think time 1) and a processor sharing server (rate 2). We seek the largest number of concurrent users the system can sustain while the server response time stays within an SLA of 2 time units. Feasibility is monotone non-increasing in the population, so bisection applies with the `max_feasible` direction (`opt_population_sizing.py`):

```
model = Network("Interactive")
think = Delay(model, "Think")
server = Queue(model, "AppServer", SchedStrategy.PS)
users = ClosedClass(model, "Users", 1, think)
think.setService(users, Exp(1.0))
server.setService(users, Exp(2.0))
model.addLink(think, server)
model.addLink(server, think)

problem = OptimizationProblem(model)
problem.add_variable(JobPopulation(users, bounds=(1, 50)))
problem.set_objective(MinimizeCost())   # pure feasibility sizing
problem.add_constraint(ResponseTimeConstraint(server, users, max_value=2.0))

result = BisectionSolver(problem, direction='max_feasible').solve()
```

Obtaining the following output:

```
Max users       : 5
Feasible        : True
LINE evaluations: 6
```

9.6 Robust sizing under workload scenarios

Capacity decisions should often hold under load uncertainty. Here the server pool of Tutorial 1 is sized so that the utilization SLA holds both under the average load ($\lambda = 3$) and under a peak-load scenario ($\lambda = 4.5$). Scenario models share the node and class names of the base model and are registered with `add_scenario`; the solver enforces every constraint on all scenarios (`opt_robust_sizing.py`):

```
base_model, queue = build(3.0)    # average load
peak_model, _ = build(4.5)        # peak load scenario

problem = OptimizationProblem(base_model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 12)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))
problem.add_scenario(peak_model)

result = BisectionSolver(problem).solve()
```

Obtaining the following output:

```
Robust servers  : 9 (6 for average load only)
Cost            : 90.0
Feasible        : True
```

The average load alone needs 6 servers; the peak scenario pushes the answer to $\lceil 4.5/0.5 \rceil = 9$.

9.7 Cost-performance tradeoff frontier

How much does tightening the utilization SLA cost? `ParetoSweep` re-solves the sizing problem for a sweep of utilization bounds (epsilon-constraint method) and reports the non-dominated cost frontier, here with each point solved exactly by bisection (`opt_pareto_frontier.py`):

```
sweep = ParetoSweep(
    problem,
    constraint_factory=lambda eps: UtilizationConstraint(queue, max_value=eps),
    epsilons=[0.3, 0.4, 0.5, 0.6, 0.75, 0.9],
    solver='bisection')
sweep.solve()
frontier = sweep.getFrontier()
```

Obtaining the following output:

```
Utilization bound -> optimal cost (servers)
  util <= 0.3  ->  100.0  (10)
  util <= 0.4  ->   80.0  (8)
  util <= 0.5  ->   60.0  (6)
  util <= 0.6  ->   50.0  (5)
  util <= 0.75 ->   40.0  (4)
```

The bound 0.9 is absent from the frontier: it requires the same 4 servers as 0.75, so its point is dominated. The frontier can be plotted with `sweep.plot(xlabel='utilization bound', ylabel='optimal cost')`, which draws only the non-dominated points as a piecewise-constant cost staircase (the epsilon-constraint cost is constant between sampled bounds, so a straight-line interpolation would misrepresent it) and marks the dominated sample points faintly.

9.8 Decomposing a multi-variable problem

Jointly choosing the number of servers and the service rate couples two variable types. The decomposition workflow splits them into blocks (servers first, then rates), solves the blocks in sequence with

the other block's values held fixed, and cycles until the full penalized objective stabilizes, i.e., block coordinate descent (`opt_decomposition.py`):

```
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))
problem.add_variable(ServiceRate(queue, jobs, bounds=(1.0, 4.0)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0},
                                   rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))

workflow = DecompositionWorkflow(problem)
workflow.auto_decompose()
workflow.set_solver_options(max_iterations=30, popsize=10, seed=42)
result = workflow.solve_sequential(max_cycles=4, tolerance=1e-3)
```

Obtaining the following output:

```
Converged       : True after 1 cycle(s)
Servers         : 10
Rate            : 2.045
Objective       : 140.90
```

As with any coordinate descent scheme on a non-convex problem, the workflow converges to a block-wise optimum that may differ from the joint optimum; a joint solve on this instance finds cost 76. Decomposition remains preferable when the joint dimensionality is intractable.

9.9 The models behind the tutorials

The tutorials above are design problems, and each one optimizes a queueing network that the narrative describes but does not draw. This section gives that network for every script of `examples/opt/`: its topology, its features, and the output the script prints once the optimizer has run.

9.9.1 `opt_bisection_sizing`

The same sizing problem solved exactly by bisection, which is admissible because the constraint is monotone in the number of servers.

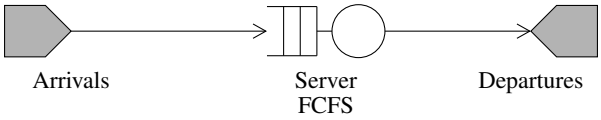

Figure 9.1: Topology of `opt_bisection_sizing`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(1)
Features	class switching on 3 links
Solvers	MVA

Table 9.1: Features of `opt_bisection_sizing`.

The example is built and solved as follows.

```
Tutorial 2: Exact sizing with the bisection solver.
Same sizing question as Tutorial 1, but solved exactly in O(log n) LINE
```

```
evaluations with BisectionSolver instead of differential evolution. This
applies whenever a single integer variable has monotone feasibility:
adding servers can only make the constraints easier to satisfy.
                      MinimizeCost, UtilizationConstraint, BisectionSolver)
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))
queue.setService(jobs, Exp(1.0))
model.addLink(source, queue)
model.addLink(queue, sink)
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))
result = BisectionSolver(problem).solve()
print(f"Optimal servers : {result.variable_values['Server_servers']}")
print(f"Cost            : {result.objective_value:.1f}")
print(f"LINE evaluations: {result.model_evaluations} (vs hundreds for DE)")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Optimal servers : 6
Cost            : 60.0
LINE evaluations: 4 (vs hundreds for DE)
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.2 `opt_decomposition`

Decomposing a multi-variable problem into blocks solved in turn, the coordinate-descent workflow and its blockwise optimum.

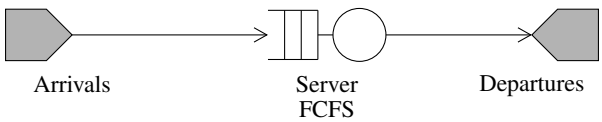

Figure 9.2: Topology of `opt_decomposition`: an open network over 1 queueing station.

The example is built and solved as follows.

```
Tutorial 8: Decomposing a multi-variable problem.
Jointly choosing the number of servers and the service rate couples two
variable types. The decomposition workflow splits them into blocks
(servers first, then rates), solves the blocks in sequence with the
```

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(1)
Features	class switching on 3 links
Solvers	MVA

Table 9.2: Features of `opt_decomposition`.

```
other block's values held fixed, and cycles until the full penalized
objective stabilizes (block coordinate descent). This scales better
than a joint search when many variable types are present, at the price
of possibly stopping in a block-wise optimum.
                      MinimizeCost, ResponseTimeConstraint,
                      DecompositionWorkflow)
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))
queue.setService(jobs, Exp(1.0))
model.addLink(source, queue)
model.addLink(queue, sink)
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))
problem.add_variable(ServiceRate(queue, jobs, bounds=(1.0, 4.0)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0},
                                   rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))
workflow = DecompositionWorkflow(problem)
workflow.auto_decompose()
workflow.set_solver_options(max_iterations=30, popsize=10, seed=42)
result = workflow.solve_sequential(max_cycles=4, tolerance=1e-3)
print(f"Converged       : {result.converged} "
      f"after {result.cycles_completed} cycle(s)")
print(f"Servers         : {result.final_variable_values['Server_servers']}")
print(f"Rate            : {result.final_variable_values['Server_Jobs_rate']:.3f}")
print(f"Objective       : {result.final_objective:.2f}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
Warning [SolverMVA]: The model has unstable queues (utilization >= 1); station utilization is reported
    capped at 1.0, queue length and response time as Inf.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
[SolverMVA] Message cast more than once, repetitions will not be printed for 60 seconds.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']

... (1050 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
```

```
Converged       : True after 2 cycle(s)
Servers         : 6
Rate            : 2.003
Objective       : 100.05
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.3 `opt_load_balancing`

Load balancing across heterogeneous servers: the decision variables are the routing probabilities, which must form a simplex.

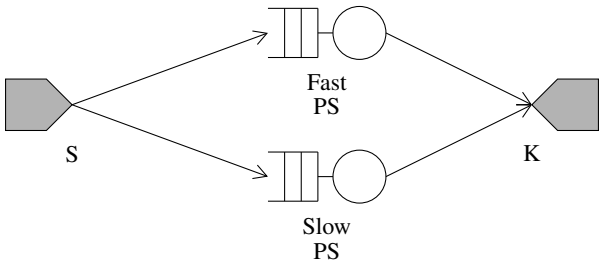

Figure 9.3: Topology of `opt_load_balancing`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: Jobs (open)
Scheduling	Fast: PS; Slow: PS
Arrivals	S – Jobs: Exp(2)
Service	Fast – Jobs: Exp(4) Slow – Jobs: Exp(2.5)
Features	class switching on 4 links
Solvers	MVA

Table 9.3: Features of `opt_load_balancing`.

The example is built and solved as follows.

```
Tutorial 4: Load balancing across heterogeneous servers.
Split an arrival stream (lambda = 2) between a fast queue (mu = 4) and a
slow queue (mu = 2.5) so that the mean end-to-end response time is
minimized. Routing probabilities are the decision variable; the system
response time is computed by Little's law from the chain queue lengths
and throughput. The exact optimum (from M/M/1 formulas) routes about 74%
of the traffic to the fast queue, not 100%: saturating the fast server
is worse than using both.
                        MinimizeSystemResponseTime)
model = Network("LoadBalance")
source = Source(model, "S")
fast = Queue(model, "Fast", SchedStrategy.PS)
slow = Queue(model, "Slow", SchedStrategy.PS)
sink = Sink(model, "K")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(2.0))
fast.setService(jobs, Exp(4.0))
slow.setService(jobs, Exp(2.5))
model.addLink(source, fast)
model.addLink(source, slow)
model.addLink(fast, sink)
model.addLink(slow, sink)
problem = OptimizationProblem(model)
problem.add_variable(RoutingProbabilities(jobs, source=source,
                                          targets=[fast, slow]))
problem.set_objective(MinimizeSystemResponseTime(jobs))
```

```
result = problem.solve(seed=42, max_iterations=60)
probs = result.variable_values['Jobs_routing_from_S']
print(f"Fraction to Fast: {probs[0]:.3f} (theory: 0.743)")
print(f"Fraction to Slow: {probs[1]:.3f}")
print(f"System RT       : {result.objective_value:.4f} (theory: 0.4264)")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (219 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
     completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Fraction to Fast: 0.736 (theory: 0.743)
Fraction to Slow: 0.264
System RT       : 0.4248 (theory: 0.4264)
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.4 `opt_lqn_host_demand`

Host demand tuning on a layered model, with the gradient optimizer and layer freezing.

Processors	3: ClientCPU (inf, $m = 1$), AppCPU (ps, $m = 1$), DbCPU (ps, $m = 1$)
Tasks	3: Client (ref, $m = 4$), AppServer (fcfs, $m = 1$), Database (fcfs, $m = 1$)
Entries	3: Browse, Render, Query
Activities	3, host demands BrowseAct: 0.5, RenderAct: 0.25, QueryAct: 0.2
Calls	2 (2 synchronous, 0 asynchronous)
Think times	Client: 1.0

Table 9.4: Features of `opt_lqn_host_demand`.

The example is built and solved as follows.

```
                    GlobalConstants, HostDemand, LayeredNetwork,
                    LineOptSolverOptions, MinimizeSystemResponseTime,
                    OptimizationProblem, Processor, SchedStrategy, Task,
                    VerboseLevel)
GlobalConstants.set_verbose(VerboseLevel.SILENT)

model = LayeredNetwork('BookstoreLQN')

P1 = Processor(model, 'ClientCPU', 1, SchedStrategy.INF)
T1 = Task(model, 'Client', 4, SchedStrategy.REF).on(P1)
T1.setThinkTime(Exp(1.0))
```

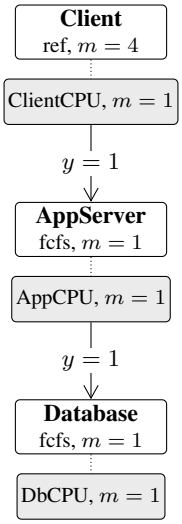

Figure 9.4: Call graph of `opt_lqn_host_demand`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
E1 = Entry(model, 'Browse').on(T1)

P2 = Processor(model, 'AppCPU', 1, SchedStrategy.PS)
T2 = Task(model, 'AppServer', 1, SchedStrategy.FCFS).on(P2)
E2 = Entry(model, 'Render').on(T2)

P3 = Processor(model, 'DbCPU', 1, SchedStrategy.PS)
T3 = Task(model, 'Database', 1, SchedStrategy.FCFS).on(P3)
E3 = Entry(model, 'Query').on(T3)

A1 = Activity(model, 'BrowseAct', Exp(2.0)).on(T1).boundTo(E1).synchCall(E2, 1)
A2 = Activity(model, 'RenderAct', Exp(4.0)).on(T2).boundTo(E2).synchCall(E3, 1)
A2.repliesTo(E2)
A3 = Activity(model, 'QueryAct', Exp(5.0)).on(T3).boundTo(E3)
A3.repliesTo(E3)

    ... (18 source lines omitted; full implementation: python/examples/opt/)

print('Render demand   : %.4f' % result.getVariableValue('RenderAct_hostdemand'))
print('Query demand    : %.4f' % result.getVariableValue('QueryAct_hostdemand'))
print('System RespT    : %.4f' % result.objectiveValue)
print('Feasible        : %d' % int(result.feasible))
print('Total violation : %.4f' % result.getTotalViolation())
print('LINE evaluations: %d' % result.modelEvaluations)
```

Running this edition prints

```
Render demand   : 0.0500
Query demand    : 0.0500
System RespT    : 0.6227
Feasible        : 1
Total violation : 0.0000
LINE evaluations: 9
```

9.9.5 `opt_pareto_frontier`

The cost-performance tradeoff frontier, swept rather than solved at one weighting, through `ParetoSweep`.
The example is built and solved as follows.

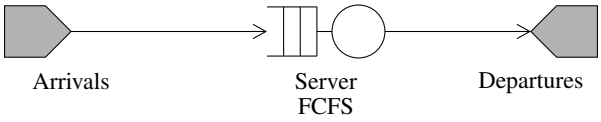

Figure 9.5: Topology of `opt_pareto_frontier`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(1)
Features	class switching on 3 links
Solvers	MVA

Table 9.5: Features of `opt_pareto_frontier`.

```
Tutorial 7: Cost-performance tradeoff frontier.
How much does tightening the utilization SLA cost? ParetoSweep re-solves
the sizing problem for a sweep of utilization bounds (epsilon-constraint
method) and reports the non-dominated cost frontier. Each point is
solved exactly with the bisection solver.
                      MinimizeCost, UtilizationConstraint, ParetoSweep)
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))
queue.setService(jobs, Exp(1.0))
model.addLink(source, queue)
model.addLink(queue, sink)
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 12)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
sweep = ParetoSweep(
    problem,
    constraint_factory=lambda eps: UtilizationConstraint(queue, max_value=eps),
    epsilons=[0.3, 0.4, 0.5, 0.6, 0.75, 0.9],
    solver='bisection',
)
sweep.solve()
print("Utilization bound -> optimal cost (servers)")

    ... (4 source lines omitted; full implementation: python/examples/opt/)

# sample points (e.g. util <= 0.9, same 4 servers as util <= 0.75) are shown
# faintly and are not part of the frontier line.
out = os.path.join(os.path.dirname(__file__), 'opt_pareto_frontier.png')
sweep.plot(xlabel='utilization bound', ylabel='optimal cost',
           title='Cost-performance Pareto frontier', save_path=out)
print(f"Frontier plot saved to {out}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
```

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (79 captured-output lines omitted; rerun the source example for the full output)

  util <= 0.3  ->  100.0  (10)
  util <= 0.4  ->   80.0  (8)
  util <= 0.5  ->   60.0  (6)
  util <= 0.6  ->   50.0  (5)
  util <= 0.75 ->   40.0  (4)
/homes/gcasale/.config/matplotlib is not a writable directory
Matplotlib created a temporary cache directory at <tmp> because there was an issue with the default path
    (/homes/gcasale/.config/matplotlib); it is highly recommended to set the MPLCONFIGDIR environment
    variable to a writable directory, in particular to speed up the import of Matplotlib and to better
    support multiprocessing.
Frontier plot saved to python/examples/opt/opt_pareto_frontier.png
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.6 opt_population_sizing

Population sizing in a closed network: how many jobs to admit so that throughput is maximized without breaching a residence-time bound.

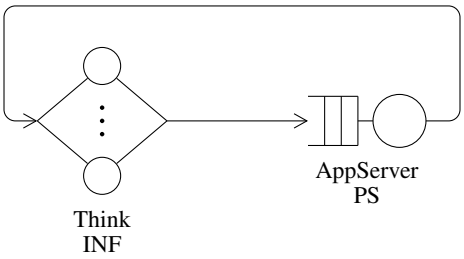

Figure 9.6: Topology of `opt_population_sizing`: a closed network over 2 queueing stations.

Nodes	2: Queue, Delay
Classes	1: Users (closed, $N = 1$)
Scheduling	Think: INF; AppServer: PS
Service	Think – Users: Exp(1) AppServer – Users: Exp(2)
Features	class switching on 2 links
Solvers	MVA

Table 9.6: Features of `opt_population_sizing`.

The example is built and solved as follows.

```
Tutorial 5: Population sizing in a closed network.
An interactive system is modeled as a closed network: a Delay station
(think time, mean 1) and a PS server (rate 2). Find the largest number
of concurrent users the system can sustain while the server response
time stays within an SLA of 2 time units. Feasibility is monotone
non-increasing in the population, so the bisection solver applies with
direction='max_feasible'.
                      ResponseTimeConstraint, BisectionSolver)
model = Network("Interactive")
think = Delay(model, "Think")
server = Queue(model, "AppServer", SchedStrategy.PS)
```

```
users = ClosedClass(model, "Users", 1, think)
think.setService(users, Exp(1.0))     # think rate 1 (mean think time 1)
server.setService(users, Exp(2.0))    # service rate 2
model.addLink(think, server)
model.addLink(server, think)
problem = OptimizationProblem(model)
problem.add_variable(JobPopulation(users, bounds=(1, 50)))
problem.set_objective(MinimizeCost())   # pure feasibility sizing
problem.add_constraint(ResponseTimeConstraint(server, users, max_value=2.0))
result = BisectionSolver(problem, direction='max_feasible').solve()
print(f"Max users       : {result.variable_values['Users_population']}")
print(f"Feasible        : {result.feasible}")
print(f"LINE evaluations: {result.model_evaluations}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'NC', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: python; env: python]
    completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'NC', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'NC', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'NC', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: exact; type: exact, deterministic; lang: python; env: python] completed in <t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (3 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'NC', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Max users       : 5
Feasible        : True
LINE evaluations: 6
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.7 `opt_robust_sizing`

Robust sizing, in which the constraint must hold across a set of workload scenarios rather than at one nominal load.

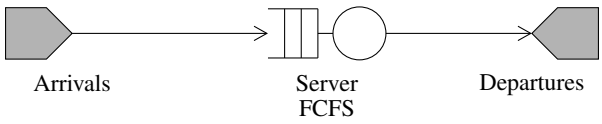

Figure 9.7: Topology of `opt_robust_sizing`: an open network over 1 queueing station.

The example is built and solved as follows.

```
Tutorial 6: Robust sizing under workload scenarios.
Size the server pool so that the utilization SLA holds both under the
average load (lambda = 3) and under a peak-load scenario (lambda = 4.5).
Scenario models share the node and class names of the base model; the
solver enforces every constraint on all scenarios. The average load
```

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(1)
Features	class switching on 3 links
Solvers	MVA

Table 9.7: Features of `opt_robust_sizing`.

```
alone needs 6 servers, but the peak scenario pushes the answer to 9.
                    MinimizeCost, UtilizationConstraint, BisectionSolver)
def build(arrival_rate):
    model = Network("MMc")
    source = Source(model, "Arrivals")
    queue = Queue(model, "Server", SchedStrategy.FCFS)
    sink = Sink(model, "Departures")
    jobs = OpenClass(model, "Jobs")
    source.setArrival(jobs, Exp(arrival_rate))
    queue.setService(jobs, Exp(1.0))
    model.addLink(source, queue)
    model.addLink(queue, sink)
    return model, queue
base_model, queue = build(3.0)        # average load
peak_model, _ = build(4.5)            # peak load scenario
problem = OptimizationProblem(base_model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 12)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))
problem.add_scenario(peak_model)
result = BisectionSolver(problem).solve()
print(f"Robust servers  : {result.variable_values['Server_servers']} "
      f"(6 for average load only)")
print(f"Cost            : {result.objective_value:.1f}")
print(f"Feasible        : {result.feasible}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (3 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Robust servers  : 9 (6 for average load only)
Cost            : 90.0
Feasible        : True
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.8 `opt_sensitivity_report`

The sensitivity of the optimum to each parameter, reported alongside the solution.

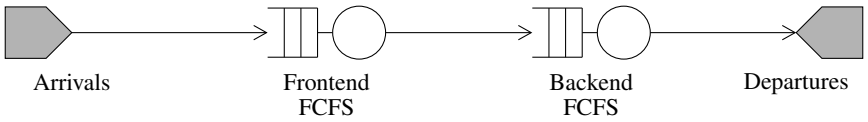

Figure 9.8: Topology of `opt_sensitivity_report`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	1: Jobs (open)
Scheduling	Frontend: FCFS; Backend: FCFS
Arrivals	Arrivals – Jobs: Exp(1)
Service	Frontend – Jobs: Exp(2) Backend – Jobs: Exp(1.5)

Table 9.8: Features of `opt_sensitivity_report`.

The example is built and solved as follows.

```
                    OpenClass, Queue, SchedStrategy, Sink, Source,
                    VerboseLevel)
"""MATLAB joins a composite key with '||'; do the same for the report."""
if isinstance(k, (tuple, list)):
    return '||'.join(str(p) for p in k)
return str(k)
GlobalConstants.set_verbose(VerboseLevel.SILENT)

model = Network('Tandem')
source = Source(model, 'Arrivals')
q1 = Queue(model, 'Frontend', SchedStrategy.FCFS)
q2 = Queue(model, 'Backend', SchedStrategy.FCFS)
sink = Sink(model, 'Departures')
jobs = OpenClass(model, 'Jobs')
source.setArrival(jobs, Exp(1.0))
q1.setService(jobs, Exp(2.0))
q2.setService(jobs, Exp(1.5))
model.link(Network.serialRouting(source, q1, q2, sink))

evaluator = LineEvaluator(model, [], [])
result = evaluator.evaluateValues({})

print('Solver used : %s' % result.solverUsed)
print('Feasible    : %d' % int(result.feasible))
print('Frontend Q  : %.4f' % result.getQueueLength('Frontend'))
print('Backend  Q  : %.4f' % result.getQueueLength('Backend'))

    ... (10 source lines omitted; full implementation: python/examples/opt/)

    print('\n=== d(%s)/d(rate) ===' % kind)
    # sort: a dict preserves insertion order, so sort for a stable report
    for mk in sorted(by_metric, key=_key):
        by_param = by_metric[mk]
        for pk in sorted(by_param, key=_key):
            print('  %-24s %-32s % .6f' % (_key(mk), _key(pk), by_param[pk]))
```

Running this edition prints

```
Solver used : MVA
Feasible    : 1
```

```
Frontend Q  : 1.0000
Backend  Q  : 2.0000

=== d(QLen)/d(rate) ===
  Backend||Jobs            rate||Backend||Jobs             -4.000000
  Frontend||Jobs           rate||Frontend||Jobs            -1.000000

=== d(RespT)/d(rate) ===
  Backend||Jobs            rate||Backend||Jobs             -4.000000
  Frontend||Jobs           rate||Frontend||Jobs            -1.000000

=== d(Util)/d(rate) ===
  Backend                  rate||Backend||Jobs             -0.444444
  Frontend                 rate||Frontend||Jobs            -0.250000

=== d(Tput)/d(rate) ===
  Backend||Jobs            rate||Backend||Jobs              0.000000
  Frontend||Jobs           rate||Frontend||Jobs             0.000000
```

9.9.9 `opt_server_sizing`

Sizing an M/M/c queue: choose the number of servers that minimizes cost subject to a response-time bound, by differential evolution over an integer decision variable.

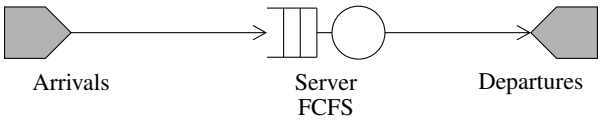

Figure 9.9: Topology of `opt_server_sizing`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(1)
Features	class switching on 3 links
Solvers	MVA

Table 9.9: Features of `opt_server_sizing`.

The example is built and solved as follows.

```
Tutorial 1: Server sizing with differential evolution.
Find the minimum number of servers of an M/M/c queue such that the
utilization does not exceed 50%. Jobs arrive at rate lambda = 3 and each
server works at rate mu = 1, so the utilization with c servers is
rho = 3/c and the optimum is c* = 6 (cost 60).
                    MinimizeCost, UtilizationConstraint)
# LINE model: Source -> Queue -> Sink
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))    # lambda = 3
queue.setService(jobs, Exp(1.0))     # mu = 1 per server
model.addLink(source, queue)
model.addLink(queue, sink)
# Optimization problem: choose 1..10 servers at 10 cost units each
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))
result = problem.solve(seed=42)
```

```
print(f"Optimal servers : {result.variable_values['Server_servers']}")
print(f"Cost            : {result.objective_value:.1f}")
print(f"Feasible        : {result.feasible}")
print(f"LINE evaluations: {result.model_evaluations}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (22 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Optimal servers : 6
Cost            : 60.0
Feasible        : True
LINE evaluations: 10
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

9.9.10 `opt_service_rate`

Continuous tuning of a service rate, the archetype of a continuous decision variable under a service-level constraint.

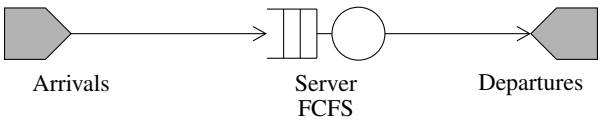

Figure 9.10: Topology of `opt_service_rate`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	1: Jobs (open)
Scheduling	Server: FCFS
Arrivals	Arrivals – Jobs: Exp(3)
Service	Server – Jobs: Exp(4)
Features	class switching on 3 links
Solvers	MVA

Table 9.10: Features of `opt_service_rate`.

The example is built and solved as follows.

```
Tutorial 3: Continuous service rate tuning.
Choose the cheapest service rate (a continuous decision) meeting a
response time SLA on an M/M/1 queue. With lambda = 3, the response time
is RT = 1/(mu - 3), so RT <= 0.5 requires mu >= 5; rate is billed at
20 cost units per unit, hence the optimum is mu* = 5 (cost 100).
                      MinimizeCost, ResponseTimeConstraint)
model = Network("MM1")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))
queue.setService(jobs, Exp(4.0))     # initial rate, will be optimized
model.addLink(source, queue)
model.addLink(queue, sink)
problem = OptimizationProblem(model)
problem.add_variable(ServiceRate(queue, jobs, bounds=(3.5, 8.0)))
problem.set_objective(MinimizeCost(rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))
result = problem.solve(seed=42, max_iterations=60)
rate = result.variable_values['Server_Jobs_rate']
print(f"Optimal rate     : {rate:.3f} (theory: 5.0)")
print(f"Cost             : {result.objective_value:.1f}")
print(f"Feasible         : {result.feasible}")
```

Running this edition prints

```
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA

... (619 captured-output lines omitted; rerun the source example for the full output)

SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
SolverAUTO: Selected solver 'MVA'
SolverAUTO: Candidates were ['MVA', 'LDES', 'Fluid', 'MAM', 'CTMC', 'SSA']
MVA analysis [method: default/exact; type: exact, deterministic; lang: python; env: python] completed in <
    t>s.
SolverAUTO: Method 'getAvgSysTable' succeeded with SolverMVA
Optimal rate     : 5.001 (theory: 5.0)
Cost             : 100.0
Feasible         : True
```

The columns are the mean queue length, utilization, response time, residence time, arrival rate and throughput of each station-class pair, as returned by `MVA`.

Index

Entries are two-level. A member of an enumerated family — a solver, a node type, a scheduling or routing strategy, a distribution, a metric, a model class, an option, a solution method — is listed both under its own name and under its family, so that `FCFS` can be reached either alphabetically or through *scheduling strategies*. Page numbers point at the definition or the explanation of a term, not at every mention of it.

`ag_closed_network`, 269
`ag_gnetwork`, 271
`ag_jackson_network`, 272
`ag_multiclass_closed`, 273
`ag_tandem_open`, 274
`ag_tandem_phasetype`, 276
agent models, 269

bisection, 380
block coordinate descent, 383
busy period, 234
`busyp_subnetwork`, 234

`Cache` node, 113
cache task, 252
`cache_compare_replc`, 114
`cache_itemsize_costcap`, 115
`cache_mmap_rr_env`, 116
`cache_replc_climb`, 117
`cache_replc_fifo`, 118
`cache_replc_hlru`, 120
`cache_replc_lru`, 121
`cache_replc_qlru`, 123
`cache_replc_routing`, 124
`cache_replc_rr`, 125
`cache_rmf_transient`, 127
`cachenet_tandem`, 128
caching, 113, 280
capacity planning, 382
`cdf_respt_closed`, 227
`cdf_respt_closed_threeclasses`, 229
`cdf_respt_distrib`, 230
`cdf_respt_open_twoclasses`, 231
`cdf_respt_populations`, 233
`cl_basic`, 139
`cl_closed`, 140
`cl_compare`, 141
`cl_multiclass`, 142
`cl_scheduling`, 143
`cl_stations`, 144
`cl_sweep`, 145
class priorities, 49
class switching, 43
`ClassSwitch` node, 43
closed queueing network, 13
cluster network, 139
constraint, 379
`cqn_bas_blocking`, 13
`cqn_bcmp_theorem`, 14
`cqn_lcfs_multiclass`, 15
`cqn_mmpp2_service`, 17
`cqn_multichain_cs`, 18
`cqn_multiserver`, 20
`cqn_multiserver_nc`, 21
`cqn_oneline`, 23
`cqn_repairmen`, 24
`cqn_repairmen_multi`, 25
`cqn_scheduling_dps`, 26
`cqn_threeclass_hyperl`, 28
`cqn_twoclass_erl`, 29
`cqn_twoclass_hyperl`, 31
`cqn_twoqueues_multi`, 32
cross-codebase parity, 1
`cs_implicit`, 43
`cs_multi_diamond`, 44
`cs_single_diamond`, 46
`cs_transient_class`, 47
`CTMC` solver, 346
`ctmc_spn_mm1`, 346
`ctmc_spn_vs_nc_closed_qn`, 347
`ctmc_tandem_mm1`, 349
custom solver, 278
cyclic polling, 220

decision variable, 379
design problem, 379, 383
differential evolution, 379
discrete time, 339

dispatch_closed, 146
dispatch_mixed, 147
dispatch_open, 148
distributions
    Geometric, 339
dt_bernoulli_loaddep, 339
dt_cycle, 340
dt_cycle_loaddep, 341
dt_cycle_multiclass, 342
dt_geogeo1, 343
dt_geogeo1_loss, 344

epsilon-constraint method, 382
est_ekf_changepoint, 358
est_ekf_closed, 359
est_ekf_open, 361
est_erps_closed, 362
est_erps_open, 363
est_mcmc_closed, 364
est_mle_open, 365
est_qmle_closed, 367
est_trace_closed, 368
est_ubo_changepoint, 369
est_ubo_closed, 370
est_ubo_open, 371
est_ubo_variants_closed, 372
est_ubr_changepoint, 374
est_ubr_closed, 375
est_ubr_open, 376
est_vi_closed, 377
example_mapqn2renv, 239
Extended Kalman Filter, 358

fcr_constraints, 187
fcr_lincon, 188
fcr_lossn, 189
fcr_mm1kdrop, 191
fcr_mm1waitq, 192
fcr_oqndrop, 193
fcr_oqnwaitq, 194
feature table, 1
fes_aggregation, 172
fes_single_class, 174
file formats
    JSON, 351
finite capacity region, 187, 280
fj_asymm, 56
fj_basic_closed, 57
fj_basic_nesting, 59
fj_basic_open, 61
fj_complex_serial, 62
fj_cs_multi_visits, 64
fj_cs_postfork, 65
fj_cs_prefork, 67
fj_deep_nesting, 68
fj_delays, 70
fj_mixed_openclosed, 71
fj_nojoin, 74
fj_route_overlap, 75
fj_serialfjs_closed, 77
fj_serialfjs_open, 78
fj_threebranches, 80
fj_tiny_closed, 82
fj_twoclasses_forked, 83
fj_variable_fanout, 85
flow-equivalent server, 172
Fork node, 56
fork-join, 56, 280

gallery_aphm1, 281
gallery_cache_lru, 282
gallery_cache_routing, 283
gallery_coxm1, 284
gallery_cqn, 285
gallery_cqn_multiclass, 285
gallery_detm1, 286
gallery_dm1, 287
gallery_erldk, 288
gallery_erlerl1, 288
gallery_erlerl1_reentrant, 289
gallery_erlm1, 290
gallery_erlm1_ps, 291
gallery_erlm1_reentrant, 291
gallery_erlm1ps, 292
gallery_fcr, 293
gallery_fj_closed, 293
gallery_fj_open, 294
gallery_fj_quorum, 295
gallery_gamm1, 297
gallery_hyperl1_feedback, 297
gallery_hyperl1_reentrant, 298
gallery_hyperlk, 299
gallery_hyphyp1_linear, 300
gallery_hyphyp1_reentrant, 300
gallery_hyphyp1_tandem, 301
gallery_hypm1, 302
gallery_hypm1_reentrant, 303
gallery_lqn_basic, 303
gallery_lqn_random, 304
gallery_lqn_workflows, 305

gallery_lukumar_reentrant, 306
gallery_mapm1, 308
gallery_mapmk, 308
gallery_mdk, 309
gallery_merl1, 310
gallery_merl1_linear, 310
gallery_merl1_reentrant, 311
gallery_merl1_tandem, 312
gallery_merlk, 313
gallery_mhyp1, 313
gallery_mhyp1_linear, 314
gallery_mhyp1_reentrant, 315
gallery_mhyp1_tandem, 316
gallery_mhypk, 316
gallery_mm1, 317
gallery_mm1_feedback, 317
gallery_mm1_linear, 318
gallery_mm1_multiclass, 319
gallery_mm1_prio, 320
gallery_mm1_ps, 321
gallery_mm1_ps_feedback, 321
gallery_mm1_ps_multiclass, 322
gallery_mm1_ps_reentrant, 323
gallery_mm1_reentrant, 324
gallery_mm1_tandem, 324
gallery_mm1_tandem_multiclass, 325
gallery_mm1k, 326
gallery_mmap1, 327
gallery_mmap1_multiclass, 327
gallery_mmapk, 328
gallery_mmk, 329
gallery_mpar1, 330
gallery_multitier, 330
gallery_multitier_storage, 331
gallery_parm1, 333
gallery_qn_random, 334
gallery_renv_breakdown, 335
gallery_repairmen, 335
gallery_replayerm1, 336
gallery_um1, 337
Geo/Geo/1 queue, 339
Geometric distribution, 339

HOL scheduling, 49

infinitesimal generator, 346
init_state_fcfs_exp, 206
init_state_fcfs_nonexp, 207
init_state_ps, 209

Join node, 56

JSON format, 351
json_balking_retrial, 351
json_classcap_droprule, 352
json_fcr_details, 353
json_hetero_servers, 354
json_join_deadline, 354
json_signal_classes, 356

Kendall notation, 280

large-scale models, 277
largescale_tbi, 277
layered queueing network, 86, 280
lcq_singlehost, 252
lcq_threehosts, 253
ld_class_dependence, 175
ld_fes_multiclass, 176
ld_fes_singleclass, 178
ld_global_dependence, 179
ld_joint_dependence, 180
ld_multiserver_fcfs, 182
ld_multiserver_ps, 183
ld_multiserver_ps_twoclasses, 184
ld_whittle_bandwidth, 186
ldes_warmstart, 210
load balancing, 381, 383
load dependence, 172
lqn_basic, 86
lqn_bpmn, 87
lqn_bpmn_trace, 90
lqn_flatph, 92
lqn_fork_open_arrival, 93
lqn_jsq, 94
lqn_moment3, 95
lqn_multi_solvers, 97
lqn_ofbiz, 98
lqn_open_arrival, 101
lqn_paramident, 102
lqn_rrobin, 102
lqn_serial, 103
lqn_server_pools, 104
lqn_setup, 105
lqn_sockshop, 107
lqn_srvnph, 109
lqn_transient, 109
lqn_twotasks, 110
lqn_workflows, 112
Lu-Kumar re-entrant line, 280

maximum likelihood, 358
MCMC, 358

mixed network, 34
model features
caching, 113, 280
class priorities, 49
class switching, 43
cyclic polling, 220
finite capacity region, 187, 280
fork-join, 56, 280
load balancing, 381, 383
load dependence, 172
random environment, 238, 280
reward models, 264
slotted time, 339
state-dependent routing, 212
switchover times, 226
warm start, 206
model gallery, 280
model interchange, 351
mqn_basic, 34
mqn_multichain_cs, 35
mqn_multiserver_fcfs, 37
mqn_multiserver_ps, 39
mqn_singleserver_fcfs, 41
mqn_singleserver_ps, 42

node types
Cache, 113
ClassSwitch, 43
Fork, 56
Join, 56
Place, 149
Transition, 149

objective function, 379
open queueing network, 2
opt_bisection_sizing, 383
opt_decomposition, 384
opt_load_balancing, 386
opt_lqn_host_demand, 387
opt_pareto_frontier, 388
opt_population_sizing, 390
opt_robust_sizing, 391
opt_sensitivity_report, 393
opt_server_sizing, 394
opt_service_rate, 395
oqn_basic, 2
oqn_cs_routing, 4
oqn_fourqueues, 5
oqn_mapt, 6
oqn_multichain_cs, 7
oqn_nhpp, 8
oqn_oneline, 9
oqn_trace_driven, 10
oqn_vsinks, 11

parameter inference, 358
Pareto frontier, 382, 383
pas_closed_tandem_fig6, 255
pas_compatibility_5class, 256
pas_cyclic_ctmc_vs_bruteforce, 258
pas_cyclic_normconst_bruteforce, 259
pas_mmk, 260
pas_nc_sampling, 261
pas_saturation, 261
pas_selfloop, 263
pass and swap, 255
Place node, 149
polling_exhaustive_det, 220
polling_exhaustive_exp, 222
polling_gated, 223
polling_klimited, 224
prio_hol_closed, 50
prio_hol_open, 51
prio_identical, 54
prio_psprio, 54

QMLE, 358

random environment, 238, 280
renv_basic, 240
renv_container_terminal, 241
renv_fourstages_repairmen, 243
renv_genqn, 244
renv_lqn_twostages, 245
renv_map_fallback, 246
renv_node_breakdown, 247
renv_rotterdam_blending, 248
renv_threestages_repairmen, 249
renv_twostages_repairmen, 250
response time distribution, 227
retrieval_chain, 130
retrieval_default, 131
retrieval_ps, 134
retrieval_routing, 135
retrieval_simple, 138
reward models, 264
rewardModel_aggregation, 264
rewardModel_mm1k, 265
rewardModel_multiclass, 267
rewardModel_templates, 268

round-trip fixture, 351
routing matrix, 1
runnable entry, 1

scheduling strategies
  HOL, 49
script structure, 1
sdroute_closed, 212
sdroute_jsq, 213
sdroute_krzesinski, 215
sdroute_multibranch, 216
sdroute_open, 217
sdroute_twoclasses_closed, 219
serial routing, 1
server sizing, 379, 383
slot, 339
slotted time, 339
snc_delay_quantile, 236
snc_tandem_multiclass, 237
solution methods
  bisection, 380
  block coordinate descent, 383
  differential evolution, 379
  epsilon-constraint method, 382
  Extended Kalman Filter, 358
  flow-equivalent server, 172
  maximum likelihood, 358
  MCMC, 358
  QMLE, 358
  stochastic network calculus, 236
  variational inference, 358
solver_custom, 278
solver_custom_analyzer, 278
solvers
  CTMC, 346
spn_basic_closed, 149
spn_basic_open, 150
spn_closed_fourplaces, 151
spn_closed_twoplaces, 152
spn_fluid_dae, 154
spn_fourmodes, 155
spn_inhibiting, 156
spn_lpbounds, 158
spn_open_sevenplaces, 159
spn_pareto_service, 160
spn_productform_nc, 161
spn_queueing_place, 163
spn_twomodes, 164
state probability, 196
state-dependent routing, 212
statepr_aggr, 196
statepr_aggr_large, 197
statepr_allprobs_fcfs, 198
statepr_allprobs_ps, 200
statepr_sys_aggr, 201
statepr_sys_aggr_large, 203
statepr_sys_marg, 204
stochastic network calculus, 236
stochastic Petri net, 149, 350
switchover times, 226
switchover_basic, 226

test_spn_nrm_open, 165
topology figure, 1
Transition node, 149

variational inference, 358

warm start, 206
wf_aph, 166
wf_branch, 167
wf_complex, 167
wf_erlang, 168
wf_loop, 169
wf_parallel, 169
wf_serial, 170
wire key, 357
worked example, 1
workflow model, 166
workload scenario, 382