LINE: Queueing Analysis Algorithms

Example Manual – Java Edition

Last revision: September 10, 2026

QORE Lab, Imperial College London

Contents

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: the `jline.examples` package tree of `jline.jar`, 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 `jline.examples.java.basic`, one `*Model` class holding the models and one `*Examples` class solving them, 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.

```
Network model = new Network("myModel");
// Block 1: nodes
Delay node1 = new Delay(model, "Delay");
Queue node2 = new Queue(model, "Queue1", SchedStrategy.FCFS);
Source node3 = new Source(model, "Source");
Sink node4 = new Sink(model, "Sink");
// Block 2: classes
OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
node1.setService(jobclass1, new HyperExp(0.5, 3.0, 10.0));
node2.setService(jobclass1, new Exp(1.0));
node3.setArrival(jobclass1, new Exp(0.1));
// Block 3: topology
RoutingMatrix P = model.initRoutingMatrix();
P.set(jobclass1, jobclass1, new Matrix("[0,1,0,0; 0,0,0,1; 1,0,0,0; 0,0,0,0]"));
model.link(P);
// Block 4: solution
new MVA(model).getAvgTable().print();
```

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.
The example is built and solved as follows.

```
public static void oqn_basic() throws Exception {
    Network model = OpenModel.oqn_basic();
```

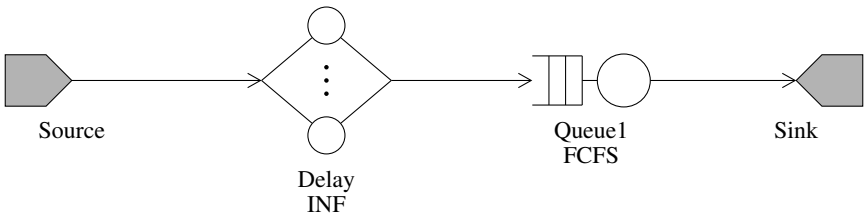

Figure 2.1: Topology of `oqn_basic`: an open network over 2 queueing stations.

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`.

```
    // Create array of different solvers to compare
    CTMC solverCTMC = new CTMC(model, "keep", false, "cutoff", 10);
    JMT solverJMT = new JMT(model, "seed", 23000, "verbose", VerboseLevel.STD, "keep", false)
    ;
    SSA solverSSA = new SSA(model, "seed", 23000, "verbose", VerboseLevel.STD, "samples",
    10000);
    FLD solverFluid = new FLD(model);
    MVA solverMVA = new MVA(model);
    NC solverNC = new NC(model);

    // Execute each solver and display results
    try {
        solverCTMC.getAvgTable().print();
    } catch (Exception e) {
        System.out.println("Solver failed: " + e.getMessage());
    }

    try {
        solverJMT.getAvgTable().print();
    } catch (Exception e) {
        System.out.println("Solver failed: " + e.getMessage());
    }

    try {
        solverSSA.getAvgTable().print();

    ... (28 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    try {
        new LDES(model, "seed", 23000, "samples", 200000).getAvgTable().print();
    } catch (Exception e) {
        System.out.println("Solver failed: " + e.getMessage());
    }
    pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 10 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
```

```
Station  JobClass      QLen      Util    RespT   ResidT      ArvR      Tput
Delay    Class1    0.022679  0.022679  0.21483  0.21483  0.099621  0.099607
Queue1   Class1     0.11332   0.10222   1.1075   1.1075  0.099607   0.10001
Source   Class1           0         0        0        0         0  0.099621
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
    >s. Iterations: 10000.
Station  JobClass      QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.021778  0.021778  0.21365  0.21365      0.1  0.10194
Queue1   Class1     0.12018   0.10537   1.1406   1.1406  0.10194  0.10537

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

LDES events:   4000    8000  12000  16000  20000  24000  28000  32000  36000  40000  44000  48000  52000
    56000  60000  64000  68000  72000  76000  80000  84000  88000  92000  96000 100000 104000 108000
    112000 116000 120000 124000 128000 132000 136000 140000 144000 148000 152000 156000 160000 164000
    168000 172000 176000 180000 184000 188000 192000 196000 200000 200000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 200000.
Station  JobClass      QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1    0.021586  0.021697  0.21538  0.21538      0.1  0.10014
Queue1   Class1     0.11144   0.10026   1.1112   1.1112  0.10014  0.10014
Source   Class1           0         0        0        0        0      0.1

[Running in non-interactive mode, continuing...]
```

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.

2.1.2 `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.

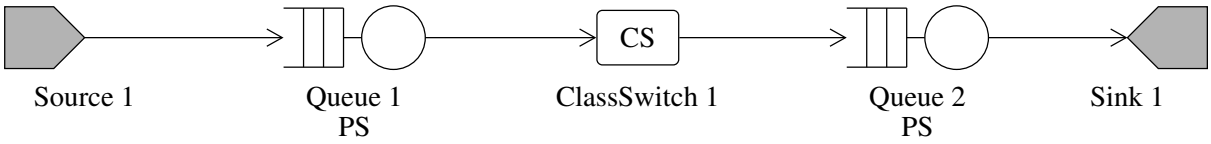

Figure 2.2: Topology of `oqn_cs_routing`: an open network over 2 queueing stations with a class-switch node.

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.

```
Network model = OpenModel.oqn_cs_routing();

// Create multiple solvers as in notebook
CTMC solverCTMC = new CTMC(model, "keep", true, "verbose", 1, "cutoff", new int[][]{{1,1,0},
    {3,3,0}, {0,0,3}});
FLD solverFluid = new FLD(model, "keep", true, "verbose", 1);
MVA solverMVA = new MVA(model, "keep", true, "verbose", 1);
MAM solverMAM = new MAM(model, "keep", true, "verbose", 1);
NC solverNC = new NC(model, "keep", true, "verbose", 1);
JMT solverJMT = new JMT(model, "keep", true, "verbose", 1, "seed", 23000, "samples", 100000);
```

```
SSA solverSSA = new SSA(model, "keep", true, "verbose", 1, "seed", 23000, "samples", 100000);
LDES solverLDES = new LDES(model, "keep", true, "verbose", 1, "seed", 23000, "samples",
    100000);

Object[] solvers = {solverCTMC, solverFluid, solverMVA, solverMAM, solverNC, solverJMT,
    solverSSA,
        solverLDES};

// Execute all solvers and collect results
for (int i = 0; i < solvers.length; i++) {
    try {
        if (solvers[i] instanceof CTMC) {
            ((CTMC) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof FLD) {
            ((FLD) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof MVA) {
            ((MVA) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof MAM) {
            ((MAM) solvers[i]).getAvgTable().print();

    ... (8 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        }
    } catch (Exception e) {
        System.out.println("Solver failed: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
Fluid analysis [method: minnormal; type: approximate, deterministic; lang: java; env: python] completed in
     <t>s.
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: java; env: python]
    completed in <t>s.
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
Queue 1   Class A   0.99361   0.4   0.4968   0.3312     2     2
Queue 1   Class B    0.7452   0.3   0.7452   0.2484     1     1
Queue 2   Class C   0.61888  0.45  0.20629  0.20629     3     3

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

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
Queue 1   Class A    1.3086  0.39838  0.65708  0.43805       2   1.9901
Queue 1   Class B   0.96972  0.29936   0.9717   0.3239       1  0.99703
Queue 2   Class C   0.81451  0.44816   0.2726   0.2726  2.9871   2.9868

[Running in non-interactive mode, continuing...]
```

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.

```
Network model = OpenModel.oqn_fourqueues();

// Create multiple solvers as in MATLAB version (some commented out in MATLAB)
CTMC solverCTMC = new CTMC(model, "seed", 23000, "cutoff", 1);
// FLD solverFluid = new FLD(model, "seed", 23000); // Commented out in MATLAB
MVA solverMVA = new MVA(model, "seed", 23000);
MAM solverMAM = new MAM(model, "seed", 23000);
JMT solverJMT = new JMT(model, "seed", 23000, "samples", 1000000);
// SSA solverSSA = new SSA(model, "seed", 23000); // Commented out in MATLAB
// NC solverNC = new NC(model, "seed", 23000); // Commented out in MATLAB

Object[] solvers = {solverCTMC, solverMVA, solverMAM, solverJMT};
String[] solverNames = {"CTMC", "MVA", "MAM", "JMT"};

// Execute all solvers and collect results
for (int i = 0; i < solvers.length; i++) {
    try {
        if (solvers[i] instanceof CTMC) {
            ((CTMC) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof MVA) {
            ((MVA) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof MAM) {
            ((MAM) solvers[i]).getAvgTable().print();
        } else if (solvers[i] instanceof JMT) {
            ((JMT) solvers[i]).getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Solver failed: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
     truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT  ResidT      ArvR      Tput
Source   Class1          0         0         0       0         0  0.078166
Source   Class2          0         0         0       0         0  0.061126
Source   Class3          0         0         0       0         0  0.059853
Queue1   Class1     0.14622  0.093799  0.46767  1.8707   0.31266   0.31266
Queue1   Class2     0.15681   0.12225  0.64133  2.5653   0.24451   0.24451
Queue1   Class3     0.16999   0.14365  0.71004  2.8402   0.23941   0.23941
Queue2   Class1     0.10897  0.085983    1.394   1.394  0.078166  0.078166
Queue2   Class2    0.095956  0.079464   1.5698  1.5698  0.061126  0.061126
Queue2   Class3     0.10304   0.08978   1.7216  1.7216  0.059853  0.059853
Queue3   Class1     0.20295   0.15633   2.5963  2.5963  0.078166  0.078166
Queue3   Class2     0.17126   0.12837   2.8018  2.8018  0.061126  0.061126
Queue3   Class3     0.15414   0.11372   2.5753  2.5753  0.059853  0.059853
Queue4   Class1     0.15103   0.11725   1.9322  1.9322  0.078166  0.078166

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

Queue3   Class1      5.9986  0.39722  28.928  28.928  0.19922  0.19922
Queue3   Class2      3.8567  0.26258  30.078  30.078  0.12464  0.12464
Queue3   Class3      3.9759  0.27255  26.901  26.901  0.14254  0.14267
Queue4   Class1      1.3583  0.29939  6.6818  6.6818  0.19939  0.19939
Queue4   Class2     0.72756   0.1126  5.7367  5.7367  0.12507   0.1246
Queue4   Class3      1.0722  0.32813  7.4165  7.4165  0.14274  0.14286

[Running in non-interactive mode, continuing...]
```

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

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.

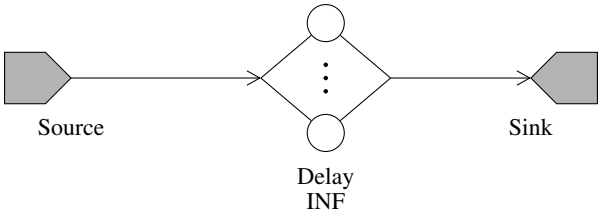

Figure 2.3: Topology of `oqn_mapt`: an open network over 1 queueing station.

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 example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.1.6 oqn_nhpp

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

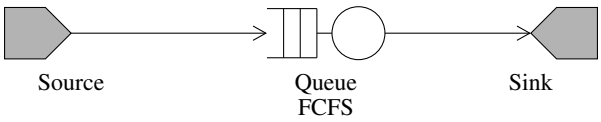

Figure 2.4: Topology of `oqn_nhpp`: an open network over 1 queueing station.

The example is built and solved as follows.

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`.

```
Network model = OpenNHPPModel.oqn_nhpp();

try {
    new LDES(model, "seed", 1234, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LDES failed: " + e.getMessage());
}
try {
    new FLD(model, "timespan", new double[]{0, 12}, "verbose", 0).getAvgTable().print();
} catch (Exception e) {
    System.out.println("FLD failed: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
LDES events:   2000    4000   6000   8000  10000  12000  14000  16000  18000  20000  22000  24000  26000
     28000  30000  32000  34000  36000  38000  40000  42000  44000  46000  48000  50000  52000  54000
     56000  58000  60000  62000  64000  66000  68000  70000  72000  74000  76000  78000  80000  82000
     84000  86000  88000  90000  92000  94000  96000  98000 100000 100000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 100000.
Station  JobClass     QLen    Util    RespT   ResidT    ArvR    Tput
Source   OpenClass       0       0        0        0       0  3.6667
Queue    OpenClass  0.7226  0.3662  0.19751  0.19751  3.6667  3.6581
Fluid analysis [method: closing; type: approximate, deterministic; lang: java; env: python] completed in <
     t>s.
Fluid analysis [method: closing; type: approximate, deterministic; lang: java; env: python] completed in <
     t>s.
Station  JobClass    QLen  Util  RespT  ResidT    ArvR    Tput
Source   OpenClass     0     0      0       0       0  3.6667
Queue    OpenClass   0.4   0.4    0.1       0  3.6667       4

[Running in non-interactive mode, continuing...]
```

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

2.1.7 `oqn_oneline`

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

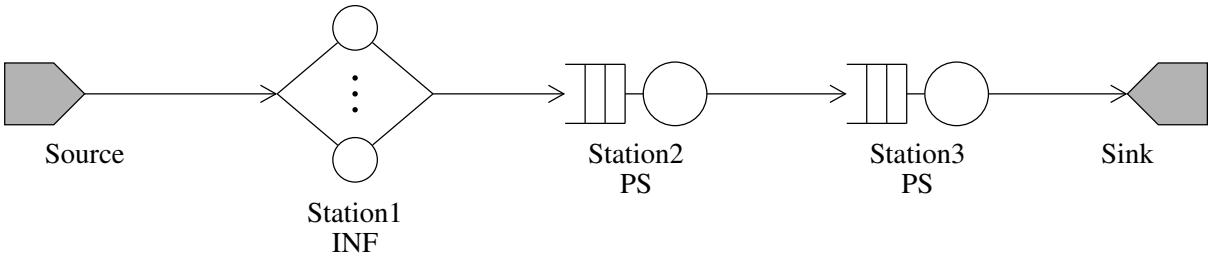

Figure 2.5: Topology of `oqn_oneline`: an open network over 3 queueing stations.

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.

```
Network model = OpenModel.oqn_oneline();

// Create solver as in MATLAB version
MVA solver = new MVA(model);
solver.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 44.
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

[Running in non-interactive mode, continuing...]
```

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`.

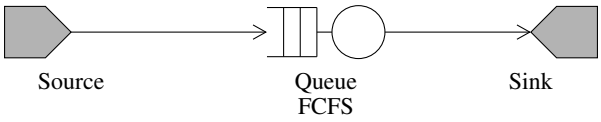

Figure 2.6: Topology of `oqn_trace_driven`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Network model = OpenModel.oqn_trace_driven();
```

```
try {
    new JMT(model, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
try {
    new LDES(model, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LDES failed: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
FAILED oqn_trace_driven: java.lang.RuntimeException: java.io.FileNotFoundException: file:common/jline.jar
    !/example_trace.txt (No such file or directory)
```

2.1.9 oqn_vsinks

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

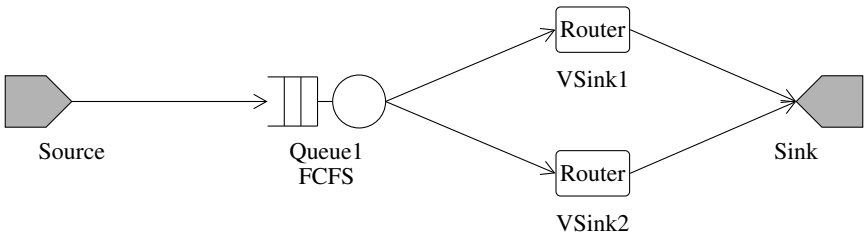

Figure 2.7: Topology of oqn_vsinks: an open network over 1 queueing station with a router.

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.

The example is built and solved as follows.

```
Network model = OpenModel.oqn_vsinks();

// Create solvers as in MATLAB version
MVA solverMVA = new MVA(model);
MAM solverMAM = new MAM(model);
NC solverNC = new NC(model);

// MVA solver results
solverMVA.getAvgTable().print();
solverMVA.getAvgNodeTable().print();

// MAM solver results
solverMAM.getAvgTable().print();
solverMAM.getAvgNodeTable().print();

// NC solver results
solverNC.getAvgTable().print();
solverNC.getAvgNodeTable().print();
```

```
pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 16.
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
Node    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
Sink    Class1          0     0         0         0     1     0
Sink    Class2          0     0         0         0     1     0
VSink1  Class1          0     0         0         0   0.6   0.6
VSink1  Class2          0     0         0         0   0.1   0.1
VSink2  Class1          0     0         0         0   0.4   0.4

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

Sink    Class1          0     0         0         0     1     0
Sink    Class2          0     0         0         0     1     0
VSink1  Class1          0     0         0         0   0.6   0.6
VSink1  Class2          0     0         0         0   0.1   0.1
VSink2  Class1          0     0         0         0   0.4   0.4
VSink2  Class2          0     0         0         0   0.9   0.9

[Running in non-interactive mode, continuing...]
```

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.

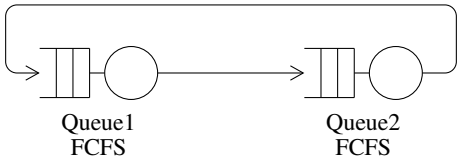

Figure 2.8: Topology of `cqn_bas_blocking`: a closed network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

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.

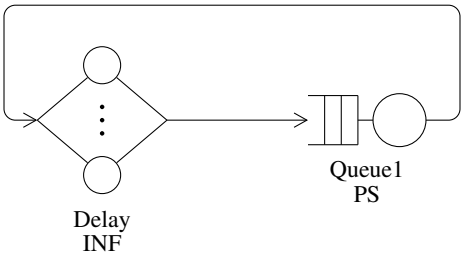

Figure 2.9: Topology of `cqn_bcmp_theorem`: a closed network over 2 queueing stations.

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.

```
// Test PS scheduling model
Network modelPS = ClosedModel.cqn_bcmp_theorem_ps();
try {
    CTMC solverPS = new CTMC(modelPS);
    solverPS.getAvgTable().print();
} catch (Exception e) {
    System.out.println("Error solving PS model: " + e.getMessage());
}

// Test FCFS scheduling model
Network modelFCFS = ClosedModel.cqn_bcmp_theorem_fcfs();
try {
    CTMC solverFCFS = new CTMC(modelFCFS);
    solverFCFS.getAvgTable().print();
} catch (Exception e) {
    System.out.println("Error solving FCFS model: " + e.getMessage());
}

// Test LCFS-PR scheduling model
Network modelLCFSPR = ClosedModel.cqn_bcmp_theorem_lcfspr();
CTMC solverLCFSPR = new CTMC(modelLCFSPR);
solverLCFSPR.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

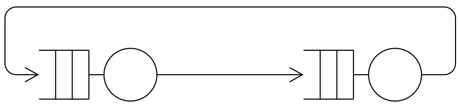

Figure 2.10: Topology of `cqn_lcfs_multiclass`: a closed network over 2 queueing stations.

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`.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.2.4 cqn_mmpp2_service

MMPP(2) service times, a correlated service process that no product-form solver can represent exactly. The example is built and solved as follows.

```
Network model = ClosedModel.cqn_mmpp2_service();

// The reference pins the seed and takes the engine's own run length, then
// runs the discrete-event engine at the same seed.
show("JMT", () -> new JMT(model, "seed", 23000).getAvgTable());
```

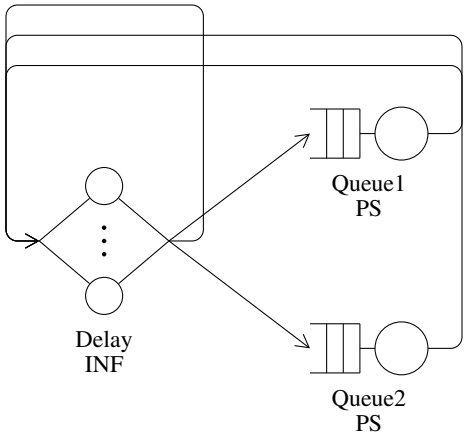

Figure 2.11: Topology of `cqn_mmpp2_service`: a closed network over 3 queueing stations.

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`.

```
show("LDES", () -> new LDES(model, "seed", 23000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
jline.lang.Network@1f89ab83
```

2.2.5 `cqn_multichain_cs`

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

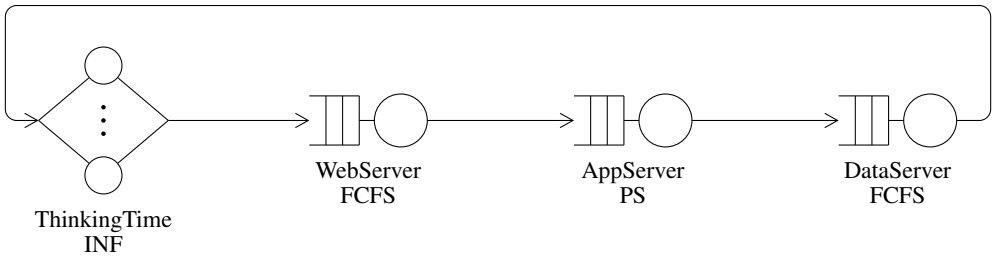

Figure 2.12: Topology of `cqn_multichain_cs`: a closed network over 4 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

2.2.6 `cqn_multiserver`

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

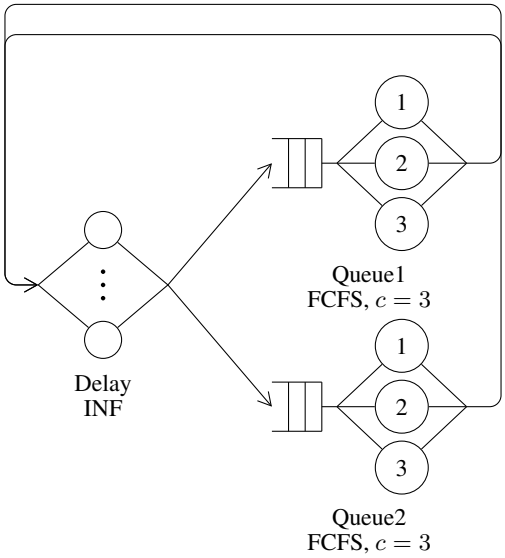

Figure 2.13: Topology of `cqn_multiserver`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
Network model = ClosedModel.cqn_multiserver();
MVA solver = new MVA(model);

solver.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/lin; type: approximate, deterministic; lang: java; env: python] completed in
     <t>s. Iterations: 124.
Station  JobClass     QLen     Util    RespT    ResidT     ArvR     Tput
Delay    Class1    0.95207  0.95207        1   0.66667  0.95207  0.95207
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
```

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`.

```
Queue2   Class3    0.67621  0.22053   2.0442    0.3407    0.3308    0.3308

[Running in non-interactive mode, continuing...]
```

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.

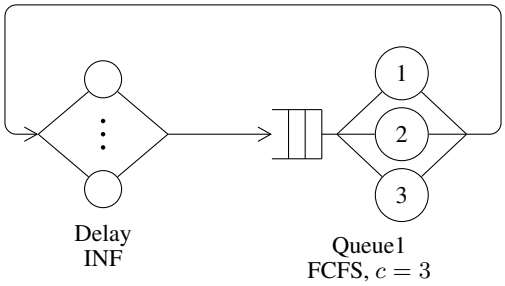

Figure 2.14: Topology of `cqn_multiserver_nc`: a closed network over 2 queueing stations.

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`.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.2.8 `cqn_oneline`

The closed counterpart of the one-line construction, from the demand and think time matrices.
The example is built and solved as follows.

```
Network model = ClosedModel.cqn_oneline();
MVA solver = new MVA(model, "method", "exact");

solver.getAvgTable().print();
```

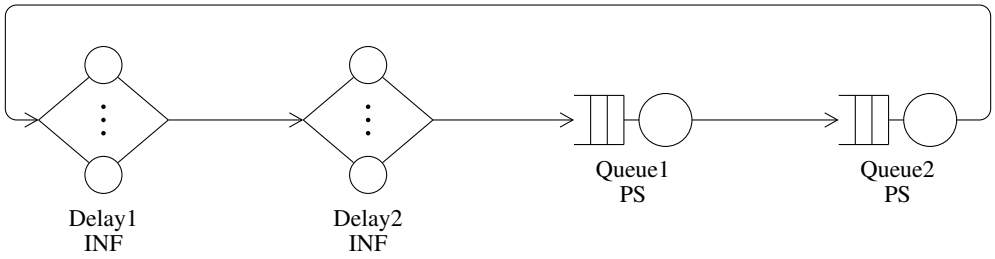

Figure 2.15: Topology of `cqn_oneline`: a closed network over 4 queueing stations.

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`.

```
pauseForUser();
```

Running this edition prints

```
MVA analysis [method: exact; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

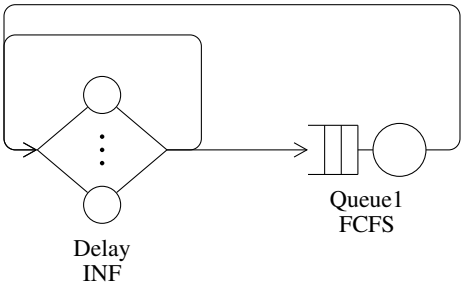

Figure 2.16: Topology of `cqn_repairmen`: a closed network over 2 queueing stations.

The example is built and solved as follows.

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`.

```
Network model = ClosedModel.cqn_repairmen();
show("CTMC", () -> new CTMC(model).getAvgTable());
show("JMT", () -> new JMT(model, "seed", 23000).getAvgTable());
show("SSA", () -> new SSA(model, "seed", 23000, "samples", 5000).getAvgTable());
show("FLD", () -> new FLD(model).getAvgTable());
show("MVA", () -> new MVA(model).getAvgTable());
show("NC", () -> new NC(model, "method", "exact").getAvgTable());
show("MAM", () -> new MAM(model).getAvgTable());
show("LDES", () -> new LDES(model, "seed", 23000, "samples", 5000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
    >s. Iterations: 5000.
Station  JobClass   QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    2.251    2.251       1       1   2.2422    2.251
Queue1   Class1    7.749  0.99977  11.626  3.4879  0.67529  0.66651
Fluid analysis [method: minnormal; type: approximate, deterministic; lang: java; env: python] completed in
     <t>s.
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: java; env: python]
    completed in <t>s.
Station  JobClass    QLen    Util   RespT  ResidT     ArvR     Tput

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

Queue1   Class1    7.778  0.99991  11.668  3.5004  0.66661  0.66661
LDES events:    100     200     300     400     500     600     700     800     900    1000    1100    1200    1300
     1400    1500    1600    1700    1800    1900    2000    2100    2200    2300    2400    2500    2600    2700
    2800    2900    3000    3100    3200    3300    3400    3500    3600    3700    3800    3900    4000    4100
    4200    4300    4400    4500    4600    4700    4800    4900    5000    5000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 5000.
Station  JobClass    QLen    Util    RespT   ResidT     ArvR     Tput
Delay    Class1    2.1531  2.1849  0.99118  0.99118   2.1885   2.1849
Queue1   Class1    7.8469       1   11.808   3.5425  0.65547  0.65909

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

```
Network model = ClosedModel.cqn_repairmen_multi();
show("CTMC", () -> new CTMC(model).getAvgTable());
// THE MULTISERVER RULE IS AN APPROXIMATION, not a setting: the golden holds
```

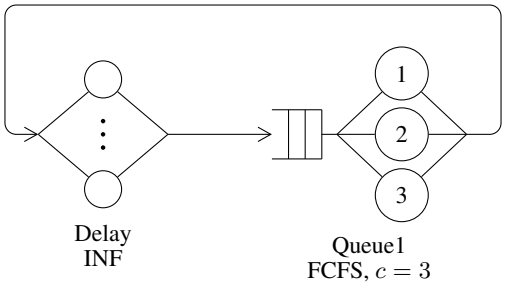

Figure 2.17: Topology of `cqn_repairmen_multi`: a closed network over 2 queueing stations.

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 softmin table because that is the one the reference prints first.
show("MVA", () -> new MVA(model, "config.multiserver", "softmin").getAvgTable());
show("MVA", () -> new MVA(model, "config.multiserver", "seidmann").getAvgTable());
show("NC", () -> new NC(model).getAvgTable());

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
MVA analysis [method: default/lin; type: approximate, deterministic; lang: java; env: python] completed in
      <t>s. Iterations: 504.
Station  JobClass     QLen      Util    RespT   ResidT    ArvR    Tput
Delay    Class1     2.4671    2.4671        1        1  2.4671  2.4671
Delay    Class2     1.3804    1.3804        1        1  1.3804  1.3804
Queue1   Class1     1.5329   0.82236  0.62136  0.62136  2.4671  2.4671
Queue1   Class2    0.61962  0.046013  0.44888  0.44888  1.3804  1.3804
MVA analysis [method: default/lin; type: approximate, deterministic; lang: java; env: python] completed in
      <t>s. Iterations: 127.
Station  JobClass    QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1    1.7794   1.7794        1        1  1.7794  1.7794
Delay    Class2    1.2993   1.2993        1        1  1.2993  1.2993
Queue1   Class1    2.2206  0.59313    1.248    1.248  1.7794  1.7794
Queue1   Class2    0.7007  0.04331  0.53929  0.53929  1.2993  1.2993
NC analysis [method: default/comom; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

2.2.11 `cqn_scheduling_dps`

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

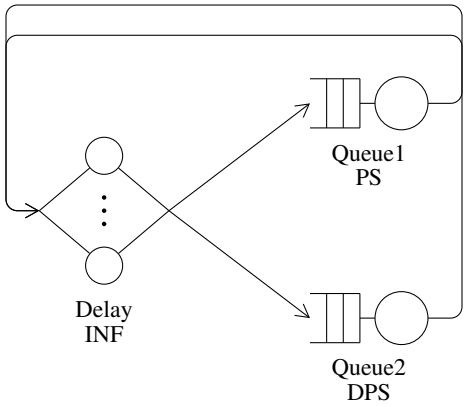

Figure 2.18: Topology of `cqn_scheduling_dps`: a closed network over 3 queueing stations.

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`.

```
Network model = ClosedModel.cqn_scheduling_dps();
show("CTMC", () -> new CTMC(model).getAvgTable());
show("JMT", () -> new JMT(model, "seed", 23000, "samples", 10000).getAvgTable());
show("FLD", () -> new FLD(model).getAvgTable());
show("MVA", () -> new MVA(model).getAvgTable());
show("LDES", () -> new LDES(model, "seed", 23000, "samples", 10000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT   ResidT      ArvR      Tput
Delay    Class1    0.037231  0.037231  0.33333  0.33333   0.11169   0.11169
Delay    Class2     0.58654   0.58654        2        2   0.29327   0.29327
Queue1   Class1     0.53462   0.33508   15.955   4.7864  0.033508  0.033508
Queue1   Class2     0.29844   0.20529   1.4537   1.0176   0.20529   0.20529
Queue2   Class1      1.4282   0.78186   18.266   12.786  0.078186  0.078186
Queue2   Class2     0.11502  0.087981   1.3074  0.39221  0.087981  0.087981
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT   ResidT      ArvR      Tput
Delay    Class1    0.037452  0.037452  0.32388  0.32388   0.11054   0.11054
Delay    Class2     0.59463   0.59463   2.0208   2.0208   0.29296   0.29168
Queue1   Class1     0.51061   0.34442   15.763   4.7288  0.033025  0.033031
Queue1   Class2     0.29395   0.19848   1.4077  0.98538   0.20436   0.20441
Queue2   Class1      1.4321   0.81316   17.991   12.594  0.078776  0.078876

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

Delay    Class1    0.036927  0.036066  0.34062  0.34062    0.1082    0.1082
Delay    Class2     0.58166   0.58427   1.9903   1.9903   0.29213   0.29213
Queue1   Class1     0.56762   0.36037   16.743   5.0229  0.032459  0.033902
Queue1   Class2     0.30045    0.2007   1.4656   1.0259   0.20449   0.20494
Queue2   Class1      1.3954   0.77226   18.692   13.084  0.075739  0.074296
Queue2   Class2     0.11789  0.090243   1.3508  0.40523   0.08764    0.0872

[Running in non-interactive mode, continuing...]
```

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.

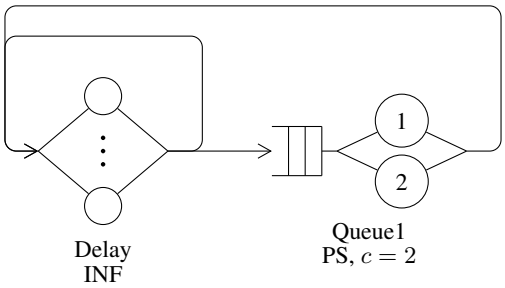

Figure 2.19: Topology of `cqn_threeclass_hyperl`: a closed network over 2 queueing stations.

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.

```
Network model = ClosedModel.cqn_threeclass_hyperl();
show("CTMC", () -> new CTMC(model).getAvgTable());
show("JMT", () -> new JMT(model, "seed", 23000, "samples", 5000).getAvgTable());
show("SSA", () -> new SSA(model, "seed", 23000, "samples", 5000).getAvgTable());
show("FLD", () -> new FLD(model).getAvgTable());
show("MVA", () -> new MVA(model).getAvgTable());
show("NC", () -> new NC(model, "method", "exact").getAvgTable());
show("MAM", () -> new MAM(model).getAvgTable());
show("LDES", () -> new LDES(model, "seed", 23000, "samples", 5000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
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
Queue1   Class2     0.60099   0.30343  0.51428   0.20816   1.1618   1.1622

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

Delay    Class1      1.1251    1.1302  0.66335   0.39485    1.668   1.6953
Delay    Class2     0.24232   0.24423  0.21479  0.086939   1.1503   1.1272
Delay    Class3     0.73684   0.71338    1.029     1.029  0.71338  0.71338
```

```
Queue1   Class1    0.032402  0.015879   0.1947  0.011589  0.16953  0.16642
Queue1   Class2     0.60015    0.2898  0.53241    0.2155   1.1272   1.1261
Queue1   Class3     0.26316   0.12628  0.36833   0.36833  0.71338  0.71338

[Running in non-interactive mode, continuing...]
```

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.

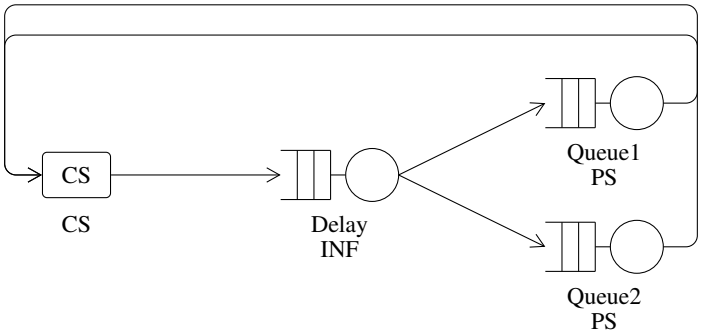

Figure 2.20: Topology of `cqn_twoclass_erl`: a closed network over 3 queueing stations with a class-switch node.

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`.

The example is built and solved as follows.

```
Network model = ClosedModel.cqn_twoclass_erl();
show("JMT", () -> new JMT(model, "seed", 23000).getAvgTable());
show("LDES", () -> new LDES(model, "seed", 23000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
LDES events:   4000    8000   12000   16000   20000   24000   28000   32000   36000   40000   44000   48000   52000
     56000   60000   64000   68000   72000   76000   80000   84000   88000   92000   96000 100000 104000 108000
     112000 116000 120000 124000 128000 132000 136000 140000 144000 148000 152000 156000 160000 164000
     168000 172000 176000 180000 184000 188000 192000 196000 200000 200000
```

```
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 200000.
Station  JobClass     QLen     Util   RespT   ResidT     ArvR     Tput
Queue1   Class1      0.9522  0.43593  3.3372  0.83431  0.28629  0.28629
Queue1   Class2     0.58817  0.28755  3.0874  0.51456  0.19086  0.19086
Queue2   Class1      7.3688  0.42701  25.739   6.4347  0.28629   0.2863
Queue2   Class2      9.9439  0.57299  26.037   8.6791  0.38172   0.3817
Delay    Class1     0.57314  0.57258  1.0008  0.50038  0.57257  0.57258
Delay    Class2     0.57374  0.57258  1.0021  0.50104  0.57259  0.57258

[Running in non-interactive mode, continuing...]
```

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.

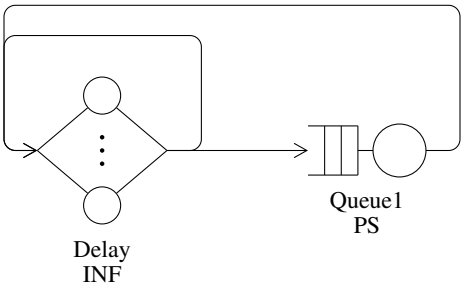

Figure 2.21: Topology of `cqn_twoclass_hyperl`: a closed network over 2 queueing stations.

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.

```
Network model = ClosedModel.cqn_twoclass_hyperl();
show("CTMC", () -> new CTMC(model).getAvgTable());
show("JMT", () -> new JMT(model, "seed", 23000, "samples", 5000).getAvgTable());
show("SSA", () -> new SSA(model, "seed", 23000, "samples", 5000).getAvgTable());
show("FLD", () -> new FLD(model).getAvgTable());
show("MVA", () -> new MVA(model, "method", "exact").getAvgTable());
show("NC", () -> new NC(model, "method", "exact").getAvgTable());
show("MAM", () -> new MAM(model).getAvgTable());
show("LDES", () -> new LDES(model, "seed", 23000, "samples", 5000).getAvgTable());

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
Queue1   Class1    0.074672  0.018632  0.54991  0.032733  0.13725  0.13724
Queue1   Class2      2.8076   0.95301   2.9302     1.186  0.95064  0.95063
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
     >s. Iterations: 5000.
Station  JobClass     QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1     1.0033    1.0033   0.6656   0.39619   1.4262   1.5073

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

LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 5000.
Station  JobClass      QLen      Util    RespT    ResidT     ArvR     Tput
Delay    Class1     0.97939   0.99866  0.65869   0.39208   1.4591    1.498
Delay    Class2     0.21006   0.21195  0.21289  0.086169   1.0217  0.97823
Queue1   Class1    0.085317  0.030225  0.60334  0.035913   0.1498  0.15358
Queue1   Class2      2.7252    0.9423   2.7906    1.1295  0.97823  0.97899

[Running in non-interactive mode, continuing...]
```

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.

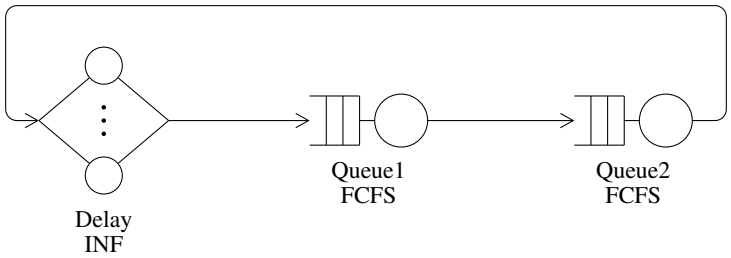

Figure 2.22: Topology of `cqn_twoqueues_multi`: a closed network over 3 queueing stations.

Nodes	3: Queue $\times$ 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`.

The example is built and solved as follows.

```
Network model = ClosedModel.cqn_twoqueues_multi();
MVA solver = new MVA(model);

solver.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

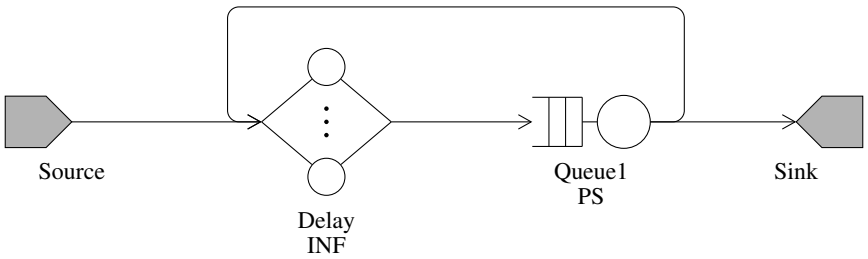

Figure 2.23: Topology of `mqn_basic`: an open network over 2 queueing stations.

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.

```
Network model = MixedModel.mqn_basic();

// Create multiple solvers for comparison as in the notebook
CTMC solverCTMC = new CTMC(model, "cutoff", 3);
JMT solverJMT = new JMT(model, "seed", 23000, "verbose", true, "keep", false);
SSA solverSSA = new SSA(model, "seed", 23000, "verbose", false, "samples", 10000);
FLD solverFluid = new FLD(model);
MVA solverMVA = new MVA(model);
NC solverNC = new NC(model);
```

```
MAM solverMAM = new MAM(model);

// Solve and display results for each solver
Object[] solvers = {solverCTMC, solverJMT, solverSSA, solverFluid, solverMVA, solverNC,
    solverMAM};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof CTMC) {
            CTMC solver = (CTMC) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof SSA) {
            SSA solver = (SSA) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof FLD) {

    ... (11 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 3 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass         QLen      Util    RespT   ResidT     ArvR     Tput
Delay    ClosedClass    1.4528    1.4528  0.66644  0.66644    2.168   2.1536
Delay    OpenClass    0.021961  0.021961  0.21581  0.21581  0.10001  0.10001
Queue1   ClosedClass   0.54717   0.40216  0.25983  0.25983   2.1536   2.1602
Queue1   OpenClass     0.16391  0.095781    1.711    1.711  0.10001  0.10001
Source   OpenClass           0         0        0        0        0  0.10001

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

Station  JobClass         QLen      Util    RespT   ResidT   ArvR   Tput
Delay    ClosedClass     1.436     1.436  0.66667  0.66667  2.154  2.154
Delay    OpenClass    0.021667  0.021667  0.21667  0.21667    0.1    0.1
Queue1   ClosedClass   0.56402   0.40925  0.26185  0.26185  2.154  2.154
Queue1   OpenClass     0.17379       0.1   1.7379   1.7379    0.1    0.1
Source   OpenClass           0         0        0        0      0    0.1

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 6 solvers the script runs (CTMC, JMT, FLD, MVA, NC, MAM), 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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

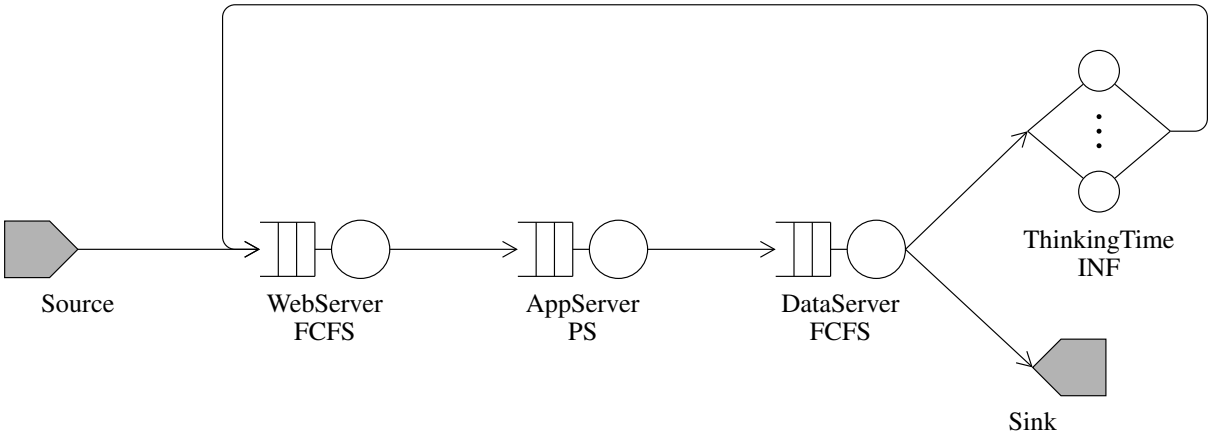

Figure 2.24: Topology of `mqn_multichain_cs`: an open network over 4 queueing stations.

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`.

2.3.3 mqn_multiserver_fcfs

The multiserver FCFS version of the same model.

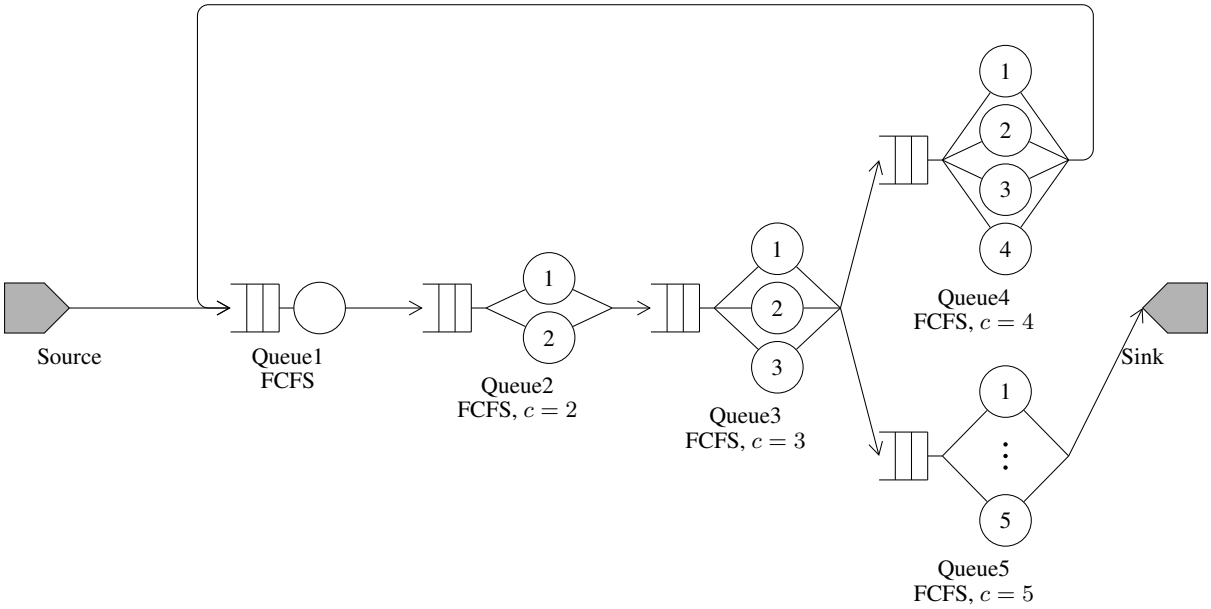

Figure 2.25: Topology of mqn_multiserver_fcfs: an open network over 5 queueing stations.

Nodes	7: Source, Queue × 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.

The example is built and solved as follows.

```
Network model = MixedModel.mqn_multiserver_fcfs();

// Create solvers with appropriate configurations as in notebook
try {
    CTMC solverCTMC = new CTMC(model, "cutoff", 3);
    solverCTMC.getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC solver not available: " + e.getMessage());
}

try {
    JMT solverJMT = new JMT(model, "samples", 100000, "seed", 23000);
    solverJMT.getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}

try {
    SSA solverSSA = new SSA(model, "cutoff", 3, "seed", 23000);
    solverSSA.getAvgTable().print();
```

```
} catch (Exception e) {
    System.out.println("SSA solver not available: " + e.getMessage());
}

try {
    MVA solverMVA = new MVA(model);

    ... (6 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    MAM solverMAM = new MAM(model, "cutoff", 3, "seed", 23000, "keep", false, "verbose", true
    );
    solverMAM.getAvgTable().print();
} catch (Exception e) {
    System.out.println("MAM solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 3 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
Queue5   OpenClass    0.11074  0.022148  0.44721  0.44721  0.24763  0.24763
Source   OpenClass          0         0        0        0        0  0.24763
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass        QLen      Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass    2.245    0.6757   3.3433   3.3433  0.67091  0.67092

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

Queue2   OpenClass    0.22745   0.10607  0.75817  0.75817      0.3      0.3
Queue3   ClosedClass  0.23504  0.078197  0.33397  0.33397  0.70378  0.70378
Queue3   OpenClass     0.1735  0.057735  0.57834  0.57834      0.3      0.3
Queue4   ClosedClass  0.17594  0.043986     0.25     0.25  0.70378  0.70378
Queue5   OpenClass    0.13416  0.026833  0.44721  0.44721      0.3      0.3
Source   OpenClass          0         0        0        0        0      0.3

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 5 solvers the script runs (CTMC, JMT, SSA, MVA, MAM), 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.

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`.

The example is built and solved as follows.

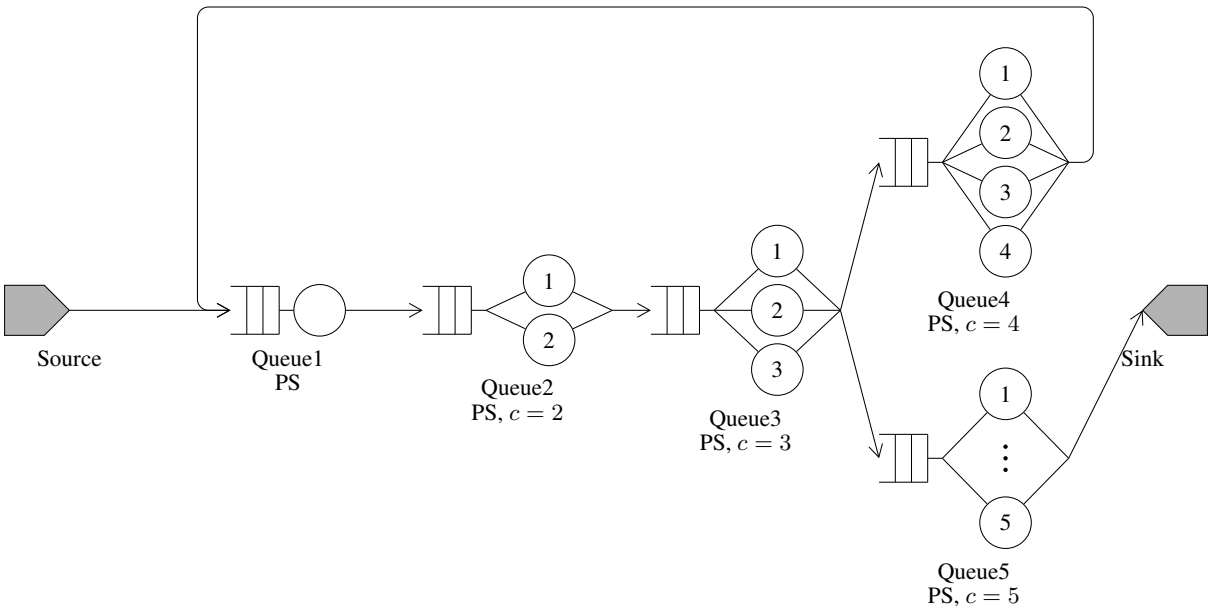

Figure 2.26: Topology of `mqn_multiserver_ps`: an open network over 5 queueing stations.

```
Network model = MixedModel.mqn_multiserver_ps();

// Create solvers with appropriate configurations as in notebook
try {
    CTMC solverCTMC = new CTMC(model, "cutoff", 3);
    solverCTMC.getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC solver not available: " + e.getMessage());
}

try {
    JMT solverJMT = new JMT(model, "samples", 100000, "seed", 23000);
    solverJMT.getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}

try {
    MVA solverMVA = new MVA(model, "method", "exact");
    solverMVA.getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA solver not available: " + e.getMessage());
}

try {
    MAM solverMAM = new MAM(model, "seed", 23000, "keep", false, "verbose", true);
    solverMAM.getAvgTable().print();
} catch (Exception e) {
    System.out.println("MAM solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 3 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass        QLen      Util    RespT    ResidT     ArvR     Tput
Queue1   ClosedClass   2.2615   0.67284   3.3567   3.3567   0.6722  0.67218

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

Queue2   OpenClass    0.22824   0.10607  0.76081  0.76081      0.3      0.3
Queue3   ClosedClass  0.22475  0.074774  0.33396  0.33396  0.67297  0.67297
Queue3   OpenClass    0.17368  0.057735  0.57894  0.57894      0.3      0.3
Queue4   ClosedClass  0.16824   0.04206     0.25     0.25  0.67297  0.67297
Queue5   OpenClass    0.13416  0.026833  0.44721  0.44721      0.3      0.3
Source   OpenClass          0         0        0        0        0      0.3

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 4 solvers the script runs (CTMC, JMT, MVA, MAM), 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.

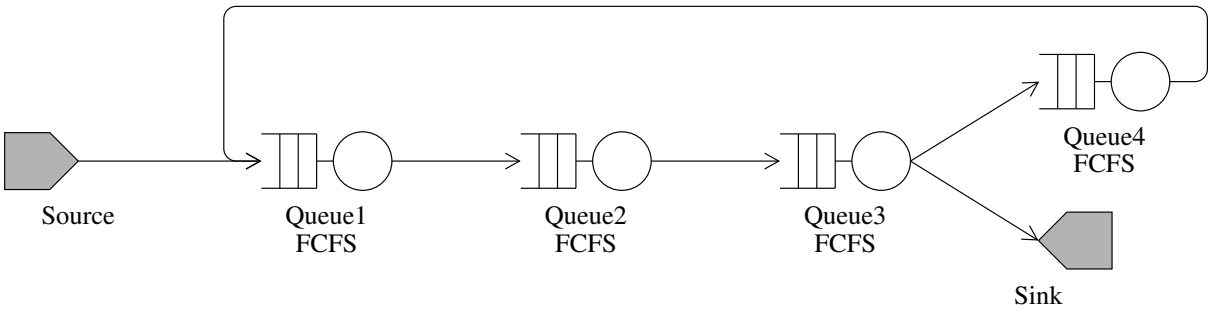

Figure 2.27: Topology of `mqn_singleserver_fcfs`: an open network over 4 queueing stations.

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.

```
Network model = MixedModel.mqn_singleserver_fcfs();

// Create solvers as in notebook - CTMC/SSA omitted due to high population
try {
    JMT solverJMT = new JMT(model, "cutoff", 3, "keep", false, "verbose", true, "seed",
    23000, "samples", 20000);
    solverJMT.getAvgTable().print();
```

```
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}

try {
    MVA solverMVA = new MVA(model, "method", "lin", "keep", false, "verbose", true, "seed",
    23000);
    solverMVA.getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA solver not available: " + e.getMessage());
}

try {
    NC solverNC = new NC(model, "keep", false, "verbose", true, "seed", 23000);
    solverNC.getAvgTable().print();
} catch (Exception e) {
    System.out.println("NC solver not available: " + e.getMessage());
}

try {
    MAM solverMAM = new MAM(model, "keep", false, "verbose", true, "seed", 23000);
    solverMAM.getAvgTable().print();
} catch (Exception e) {
    System.out.println("MAM solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: lin; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 232.
Station  JobClass        QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass   98.533  0.66435   148.31   148.31  0.66435  0.66435
Queue1   OpenClass     49.766  0.33333    149.3    149.3  0.33333  0.33333
Queue2   ClosedClass  0.83883  0.33218   1.2626   1.2626  0.66435  0.66435

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

Queue2   ClosedClass  0.99028  0.30339   1.6321   1.6321  0.60677  0.60677
Queue2   OpenClass    0.90071   0.2357   2.7021   2.7021  0.33333  0.33333
Queue3   ClosedClass  0.43798  0.20226  0.72182  0.72182  0.60677  0.60677
Queue3   OpenClass        0.42  0.19245     1.26     1.26  0.33333  0.33333
Queue4   ClosedClass  0.17882  0.15169  0.29471  0.29471  0.60677  0.60677
Source   OpenClass          0        0        0        0        0  0.33333

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 4 solvers the script runs (JMT, MVA, NC, MAM), 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.

```
Network model = MixedModel.mqn_singleserver_ps();
```

```
// Create solvers as in notebook - comprehensive comparison
try {
    JMT solverJMT = new JMT(model, "cutoff", 3, "keep", false, "verbose", true, "seed",
    23000, "samples", 100000);
    solverJMT.getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}

try {
    SSA solverSSA = new SSA(model, "seed", 23000, "verbose", false, "samples", 100000);
    solverSSA.getAvgTable().print();
} catch (Exception e) {
    System.out.println("SSA solver not available: " + e.getMessage());
}

try {
    FLD solverFluid = new FLD(model, "cutoff", 3, "keep", false, "verbose", true, "seed",
    23000, "samples", 100000);
    solverFluid.getAvgTable().print();
} catch (Exception e) {
    System.out.println("Fluid solver not available: " + e.getMessage());
}

try {
    MVA solverMVA = new MVA(model, "method", "lin");

    ... (19 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    LDES solverLDES = new LDES(model, "cutoff", 3, "keep", false, "verbose", true, "seed",
    23000, "samples", 100000);
    solverLDES.getAvgTable().print();
} catch (Exception e) {
    System.out.println("LDES solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass        QLen     Util    RespT   ResidT     ArvR     Tput
Queue1   ClosedClass   98.568  0.69772   141.74   141.74  0.70515  0.69804
Queue1   OpenClass      43.069   0.3019   143.85   143.85  0.30007  0.29886
Queue2   ClosedClass  0.82325  0.35985   1.1341   1.1341  0.69804  0.69752
Queue2   OpenClass    0.48659  0.20797    1.618    1.618  0.29886  0.29886
Queue3   ClosedClass  0.39465  0.23505  0.55513  0.55513  0.69752  0.69751
Queue3   OpenClass    0.28616   0.1685  0.96528  0.96528  0.29886  0.29905
Queue4   ClosedClass  0.21695  0.17845  0.30292  0.30292  0.69751  0.69752
Source   OpenClass          0        0        0        0        0  0.30007
Fluid analysis [method: dae; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
Fluid analysis [method: default/dae; type: approximate, deterministic; lang: java; env: python] completed
    in <t>s.
Station  JobClass        QLen     Util    RespT   ResidT  ArvR  Tput
Queue1   ClosedClass   98.924      0.7   141.32   141.32   0.7   0.7
Queue1   OpenClass     42.396      0.3   141.32   141.32   0.3   0.3

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

Queue2   ClosedClass  0.82754  0.35638   1.1601   1.1601  0.71161  0.71161
Queue2   OpenClass    0.48802  0.20879   1.6363   1.6363   0.2979   0.2979
Queue3   ClosedClass  0.39489  0.23382  0.55368  0.55368  0.71161  0.71161
Queue3   OpenClass    0.28607  0.17041  0.95876  0.95876   0.2979   0.2979
Queue4   ClosedClass    0.217  0.17767  0.30493  0.30493  0.71161  0.71165
Source   OpenClass          0        0        0        0        0      0.3

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 6 solvers the script runs (JMT, FLD, MVA, NC, MAM, LDES), which is the point of the example: the same model read by methods of different cost and different exactness.

2.4 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.

```
public static void cs_implicit() throws Exception {
    Network model = ClassSwitchingModel.cs_implicit();
    MVA solver = new MVA(model);

    solver.getAvgChainTable().print();
    pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 62.
Station   Chain   JobClasses       QLen  Util  RespT  ResidT  ArvR  Tput
Source 1  Chain1  (Class1 Class2)     0     0      0       0     0   0.6
Queue 1   Chain1  (Class1 Class2)   1.5   0.6    2.5     2.5   0.6   0.6

[Running in non-interactive mode, continuing...]
```

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.

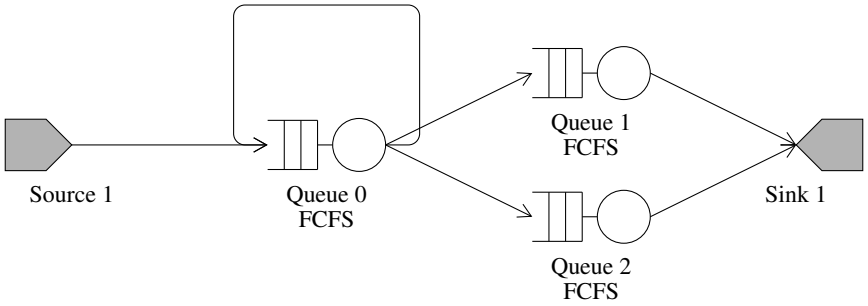

Figure 2.29: Topology of `cs_multi_diamond`: an open network over 3 queueing stations.

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`.

```
public static void cs_multi_diamond() throws Exception {
    Network model = ClassSwitchingModel.cs_multi_diamond();
    MVA solver = new MVA(model);

    solver.getAvgChainTable().print();
    pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 23.
Station   Chain   JobClasses                 QLen      Util     RespT    ResidT   ArvR   Tput
Source 1  Chain1  (Class1 Class2 Class3)         0         0         0         0      0      1
Queue 0   Chain1  (Class1 Class2 Class3)   0.14286     0.125   0.11429   0.14286   1.25   1.25
Queue 1   Chain1  (Class1 Class2 Class3)  0.019108   0.01875  0.050955  0.019108  0.375  0.375
Queue 2   Chain1  (Class1 Class2 Class3)  0.021277  0.020833  0.034043  0.021277  0.625  0.625

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

```
public static void cs_single_diamond() throws Exception {
    Network model = ClassSwitchingModel.cs_single_diamond();
    MVA solver = new MVA(model);

    solver.getAvgChainTable().print();
    pauseForUser();
```

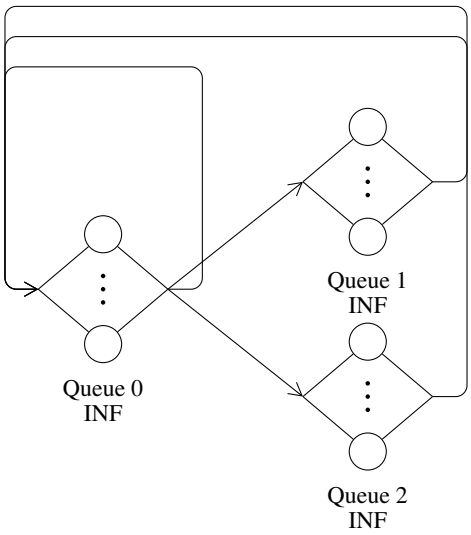

Figure 2.30: Topology of `cs_single_diamond`: a closed network over 3 queueing stations.

Nodes	3: Delay $\times$ 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`.

Running this edition prints

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

[Running in non-interactive mode, continuing...]
```

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.

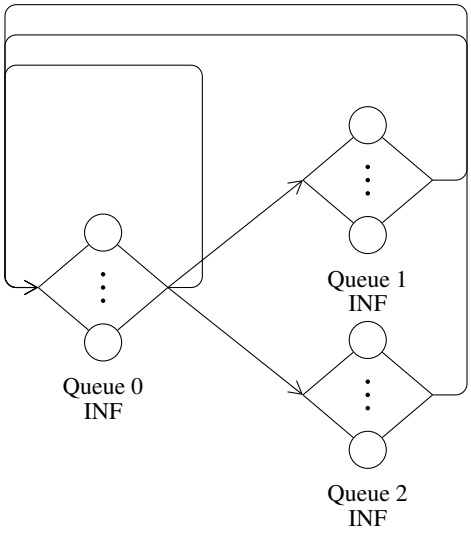

Figure 2.31: Topology of `cs_transient_class`: a closed network over 3 queueing stations.

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`.

The example is built and solved as follows.

```
public static void cs_transient_class() throws Exception {
    Network model = ClassSwitchingModel.cs_transient_class();
    model.printRoutingMatrix();

    jline.lang.NetworkStruct sn = model.getStruct(true);
    System.out.println("\n=== rt (station routing matrix) ===");
    sn.rt.print();

    // Debug SCC
    int K = sn.nclasses;
```

```
    int M = sn.nstateful;
    jline.util.matrix.Matrix inchain = sn.inchain.get(0);
    java.util.List<Integer> cols = new java.util.ArrayList<>();
    for (int ist = 0; ist < M; ist++) {
        for (int ik = 0; ik < inchain.length(); ik++) {
            cols.add(ist * K + (int) inchain.get(ik));
        }
    }
    jline.util.matrix.Matrix Pchain = new jline.util.matrix.Matrix(cols.size(), cols.size());
    for (int row = 0; row < cols.size(); row++) {
        for (int col = 0; col < cols.size(); col++) {
            Pchain.set(row, col, sn.rt.get(cols.get(row), cols.get(col)));
        }
    }
    jline.util.graph.DirectedGraph graph = new jline.util.graph.DirectedGraph(Pchain);
    jline.util.graph.DirectedGraph.SCCResult sccResult = graph.stronglyconncomp();

    ... (15 source lines omitted; full implementation: jar/src/main/java/jline/examples/)


    System.out.println("\n=== Per-Class Table ===");
    solver.getAvgTable().print();
    System.out.println("\n=== Chain Table ===");
    solver.getAvgChainTable().print();
    pauseForUser();
```

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 [Class2] => Queue 1 [Class2] : Pr=1.000000
CS_Queue 0_to_Queue 2 [Class3] => Queue 2 [Class3] : Pr=1.000000

=== rt (station routing matrix) ===
[0.2    0    0  0  0.3  0  0  0  0.5
   0    0    0  0  1.0  0  0  0    0
   0    0    0  0    0  0  0  0  1.0
 1.0    0    0  0    0  0  0  0    0
   0  1.0    0  0    0  0  0  0    0

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


=== Chain Table ===
Station  Chain   JobClasses               QLen    Util  RespT  ResidT    ArvR    Tput
Queue 0  Chain1  (Class1 Class2 Class3)    0.5     0.5      1       1     0.5     0.5
Queue 1  Chain1  (Class1 Class2 Class3)  0.1875  0.1875      1   0.375  0.1875  0.1875
Queue 2  Chain1  (Class1 Class2 Class3)  0.3125  0.3125      1   0.625  0.3125  0.3125

[Running in non-interactive mode, continuing...]
```

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.

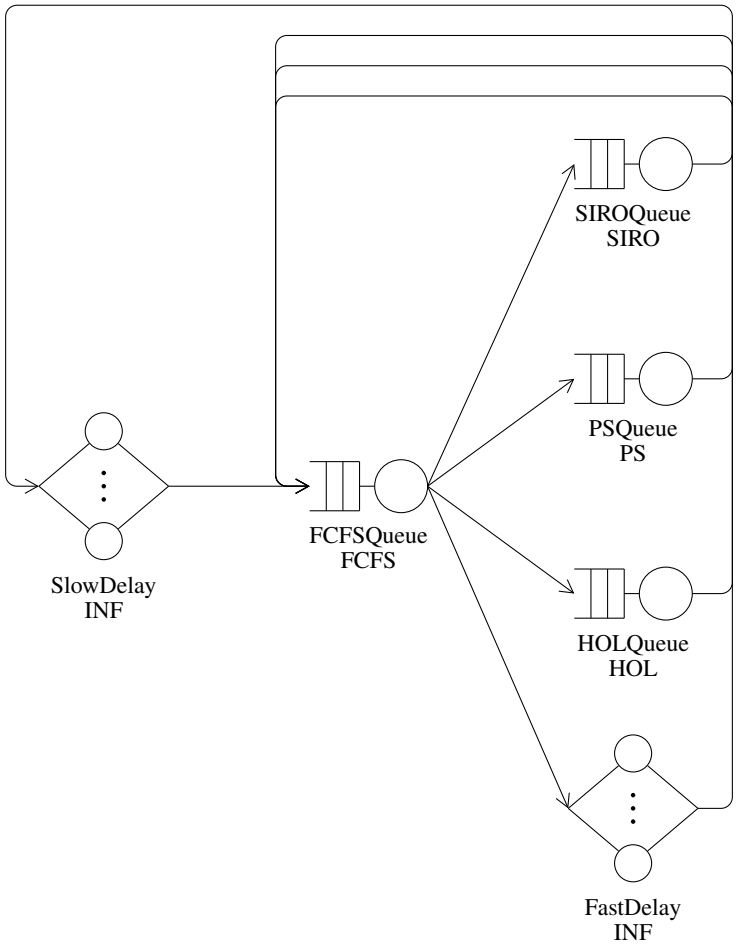

Figure 2.32: Topology of `prio_hol_closed`: a closed network over 6 queueing stations.

The example is built and solved as follows.

```
Network model = PrioModel.prio_hol_closed();

// Create solvers as in MATLAB version (lines 82, 84)
MVA solverMVA = new MVA(model, "seed", 23000, "cutoff", 1, "samples", 10000);
JMT solverJMT = new JMT(model, "seed", 23000, "cutoff", 1, "samples", 10000);

solverMVA.getAvgTable().print();
solverJMT.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
Priority: highest=Class1,Class3, lowest=Class2
Priority: highest=Class1,Class3, lowest=Class2
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 102.
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
```

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`.

```
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
PSQueue    Class2     0.0014773  0.00085805  1.8939  1.8939  0.00078005  0.00078005
PSQueue    Class3       0.46625     0.27493  3.2222  3.2222      0.1447      0.1447

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

PSQueue    Class3     0.42896  0.28248   3.1486   3.1486  0.14179  0.14205
HOLQueue   Class1      15.239   0.3766    103.3    103.3  0.14672  0.14744
HOLQueue   Class2           0        0        0        0  0.17254        0
HOLQueue   Class3      14.951   0.6199   103.84   103.84   0.1435  0.14281
FastDelay  Class1     0.14942  0.14942  0.99582  0.99582  0.14871  0.14872
FastDelay  Class3      0.1443   0.1443   0.9783   0.9783  0.14411  0.14394

[Running in non-interactive mode, continuing...]
```

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.

2.5.2 `prio_hol_open`

Head-of-line non-preemptive priority in an open network, with PS, SIRO and FCFS stations alongside.

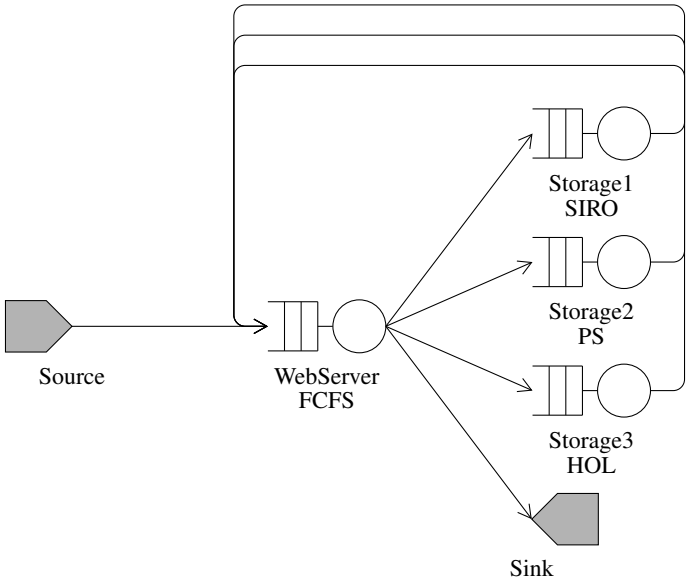

Figure 2.33: Topology of `prio_hol_open`: an open network over 4 queueing stations.

The example is built and solved as follows.

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`.

```
Network model = PrioModel.prio_hol_open();

// Create solvers as in MATLAB version (lines 69, 71, 73, 74)
CTMC solverCTMC = new CTMC(model, "seed", 23000, "cutoff", 1, "samples", 10000);
MVA solverMVA = new MVA(model, "seed", 23000, "cutoff", 1, "samples", 10000);
JMT solverJMT = new JMT(model, "seed", 23000, "cutoff", 1, "samples", 10000);
SSA solverSSA = new SSA(model, "seed", 23000, "cutoff", 1, "samples", 10000);

solverCTMC.getAvgTable().print();
solverMVA.getAvgTable().print();
solverJMT.getAvgTable().print();
solverSSA.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
Priority: highest=Class1,Class3, lowest=Class2
Priority: highest=Class1,Class3, lowest=Class2
Priority: highest=Class1,Class3, lowest=Class2
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
Storage1   Class1     0.06951  0.057509   1.3295  1.3295  0.052281  0.052281
Storage1   Class2    0.075291  0.065583   1.4924  1.4924  0.050449  0.050449
Storage1   Class3    0.076531  0.069123   1.6607  1.6607  0.046082  0.046082
Storage2   Class1     0.12865   0.10456   2.4607  2.4607  0.052281  0.052281

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

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

[Running in non-interactive mode, continuing...]
```

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.

```
Network model = PrioModel.prio_identical();

// Create solvers as in MATLAB version (lines 35, 36)
CTMC solverCTMC = new CTMC(model);
JMT solverJMT = new JMT(model, "seed", 23000, "verbose", true, "samples", 10000,
        "keep", true);

solverCTMC.getAvgTable().print();
solverJMT.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
Priority: highest=Class1, lowest=Class4
Priority: highest=Class1, lowest=Class4
Priority: highest=Class1, lowest=Class4
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass       QLen       Util     RespT    ResidT       ArvR       Tput
Delay    Class1       5.6372     5.6372   0.66667   0.66667     8.4558     8.4558
Delay    Class2      0.91384    0.91384         1         1    0.91384    0.91384
Delay    Class3       3.0493     3.0493         1         1     3.0493     3.0493
Delay    Class4    0.0031954  0.0031954       0.5       0.5  0.0063909  0.0063909
Queue1   Class1       0.3628    0.28186  0.042905  0.042905     8.4558     8.4558
Queue1   Class2       3.0862    0.45692    3.3772    3.3772    0.91384    0.91384
Queue1   Class3       0.9507    0.25411   0.31177   0.31177     3.0493     3.0493
Queue1   Class4       0.9968  0.0063909    155.97    155.97  0.0063909  0.0063909
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass       QLen       Util     RespT    ResidT       ArvR       Tput
Delay    Class1       5.6487     5.6487   0.66214   0.66214     8.5567     8.5533
Delay    Class2      0.89602    0.89602   0.99227   0.99227    0.89476    0.90847
Delay    Class3       3.0308     3.0308   0.97343   0.97343     3.1087     3.0983
Delay    Class4    0.0031932  0.0031932   0.50558   0.50558  0.0065312  0.0065312
Queue1   Class1      0.35125    0.28302  0.040881  0.040881     8.5533     8.5531
Queue1   Class2        3.104    0.42925     3.417     3.417    0.90847    0.90646
Queue1   Class3      0.96924    0.27143   0.30885   0.30885     3.0983     3.0462
Queue1   Class4      0.99678  0.0065642    152.63    152.63  0.0065312  0.0065303

[Running in non-interactive mode, continuing...]
```

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

2.5.4 `prio_psprio`

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

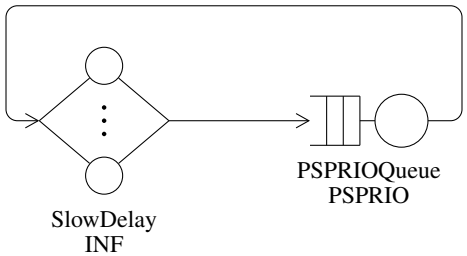

Figure 2.34: Topology of `prio_psprio`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
Network model = PrioModel.prio_psprio();
```

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`.

```
// Create solvers as in MATLAB version (lines 27, 28, 29)
CTMC solverCTMC = new CTMC(model);
JMT solverJMT = new JMT(model, "seed", 23000, "verbose", true, "samples", 5000);
SSA solverSSA = new SSA(model, "seed", 23000, "verbose", true, "samples", 5000);

solverCTMC.getAvgTable().print();
solverJMT.getAvgTable().print();
solverSSA.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
Priority: highest=Class1, lowest=Class2
Priority: highest=Class1, lowest=Class2
Priority: highest=Class1, lowest=Class2
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station      JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
SlowDelay    Class1     1.5161   1.5161   0.66563  0.66563   2.2873   2.2388
SlowDelay    Class2    0.12717  0.12717  0.20933  0.20933  0.58227  0.58227
PSPRIOQueue  Class1    0.48386  0.40341  0.21719  0.21719   2.2388   2.2385
PSPRIOQueue  Class2       1.87  0.59305    3.179    3.179  0.58227  0.57109
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
    >s. Iterations: 5000.
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

[Running in non-interactive mode, continuing...]
```

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.

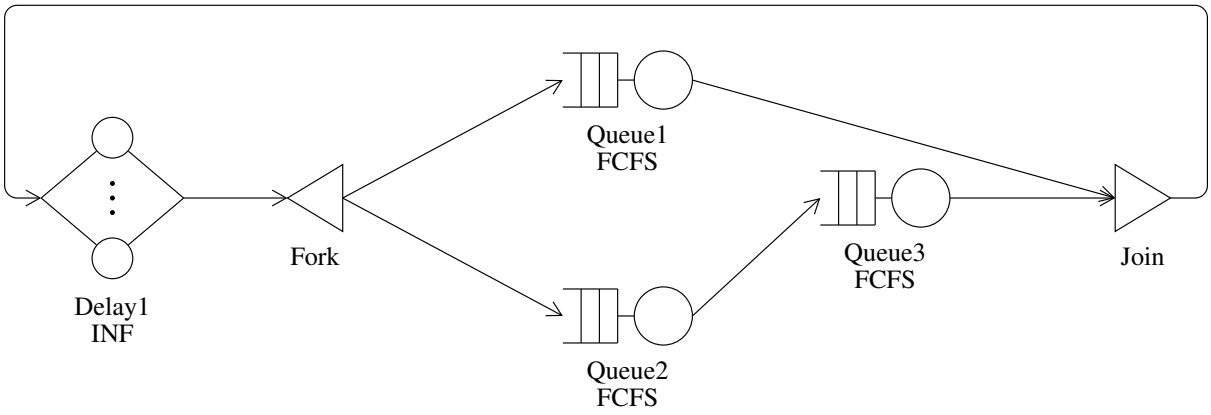

Figure 2.35: Topology of `fj_asymm`: a closed network over 4 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_asymm();

// Solve with multiple methods as in notebook
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (int i = 0; i < solvers.length; i++) {
    try {
        if (solvers[i] instanceof JMT) {
            JMT solver = (JMT) solvers[i];
            solver.getAvgTable().print();
        } else if (solvers[i] instanceof MVA) {
            MVA solver = (MVA) solvers[i];
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 673.
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

[Running in non-interactive mode, continuing...]
```

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.

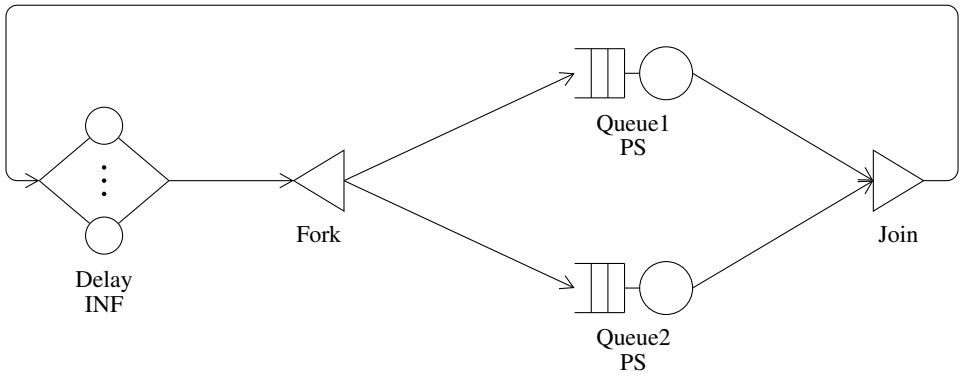

Figure 2.36: Topology of `fj_basic_closed`: a closed network over 3 queueing stations with a fork-join block.

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`.

The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_basic_closed();

// Solve with multiple methods as in notebook
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model, "fork_join", "ht", "method", "amva");

Object[] solvers = {solverJMT, solverMVA};

for (int i = 0; i < solvers.length; i++) {
    try {
        if (solvers[i] instanceof JMT) {
            JMT solver = (JMT) solvers[i];
            solver.getAvgTable().print();
        } else if (solvers[i] instanceof MVA) {
            MVA solver = (MVA) solvers[i];
            solver.getAvgTable().print();
        }
```

```
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: egflin; type: approximate, deterministic; lang: java; env: python] completed in <t>s
     . Iterations: 489.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    class1    0.77057  0.77057       1       1  0.77057  0.77057
Queue1   class1     2.8911  0.77057  3.7519  3.7519  0.77057  0.77057
Queue2   class1     2.8911  0.77057  3.7519  3.7519  0.77057  0.77057
Join     class1     2.8911        0   1.876   1.876   1.5411  0.77057

[Running in non-interactive mode, continuing...]
```

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.

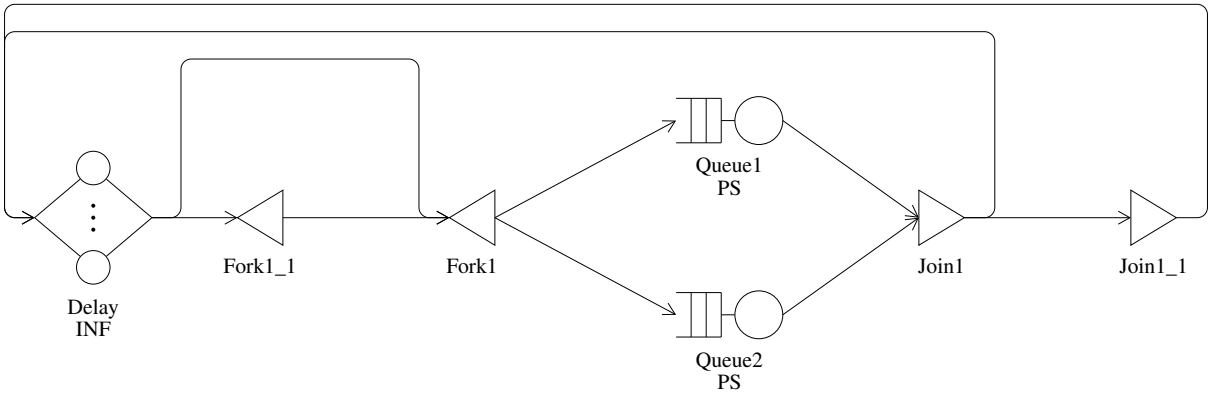

Figure 2.37: Topology of `fj_basic_nesting`: a closed network over 3 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_basic_nesting();
JMT solver = new JMT(model, "seed", 23000);
```

```
solver.getAvgTable().print();
MVA solverMVA = new MVA(model);
solverMVA.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
WARNING: [setTasksPerLink] The setTasksPerLink feature is experimental and results may be inaccurate for
     analytical solvers.
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 1558.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    class1      1.762    1.762        4        4  0.44051  0.44051
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

[Running in non-interactive mode, continuing...]
```

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.

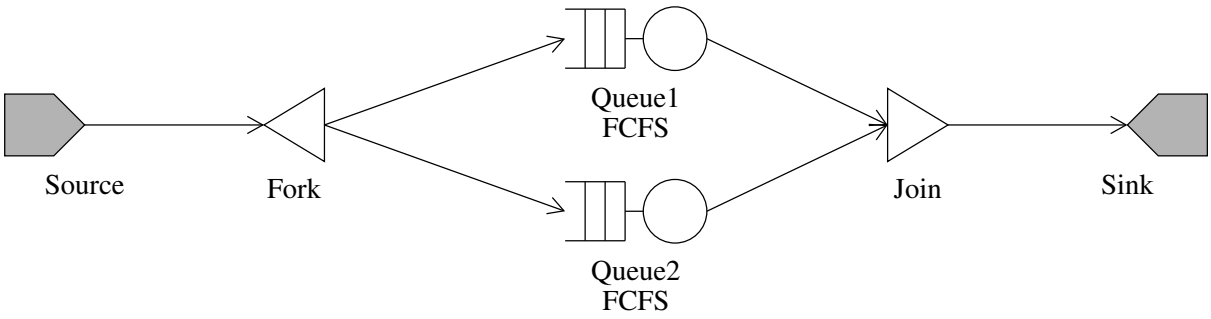

Figure 2.38: Topology of `fj_basic_open`: an open network over 2 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_basic_open();

// The solvers the reference runs, in its order. MAM is pinned to the
// source-decomposition MMAP analyzer: the default analyzer answers a
// different question on a forked open network.
try {
    new JMT(model, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
try {
    new MVA(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new LDES(model, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LDES failed: " + e.getMessage());
}
try {
    new MAM(model, "method", "dec.source.mmap").getAvgTable().print();
} catch (Exception e) {
    System.out.println("MAM failed: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 396.
Station  JobClass      QLen    Util    RespT   ResidT  ArvR  Tput
Source   class1            0       0        0        0     0  0.05
Queue1   class1     0.052631    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     0.043789       0  0.43789  0.43789   0.1  0.05
LDES events:   4000    8000  12000  16000  20000  24000  28000  32000  36000  40000  44000  48000  52000
     56000  60000  64000  68000  72000  76000  80000  84000  88000  92000  96000 100000 104000 108000
     112000 116000 120000 124000 128000 132000 136000 140000 144000 148000 152000 156000 160000 164000
     168000 172000 176000 180000 184000 188000 192000 196000 200000 200000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 200000.
Station  JobClass      QLen      Util    RespT   ResidT     ArvR      Tput

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

MAM analysis [method: dec.source.mmap; type: approximate, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

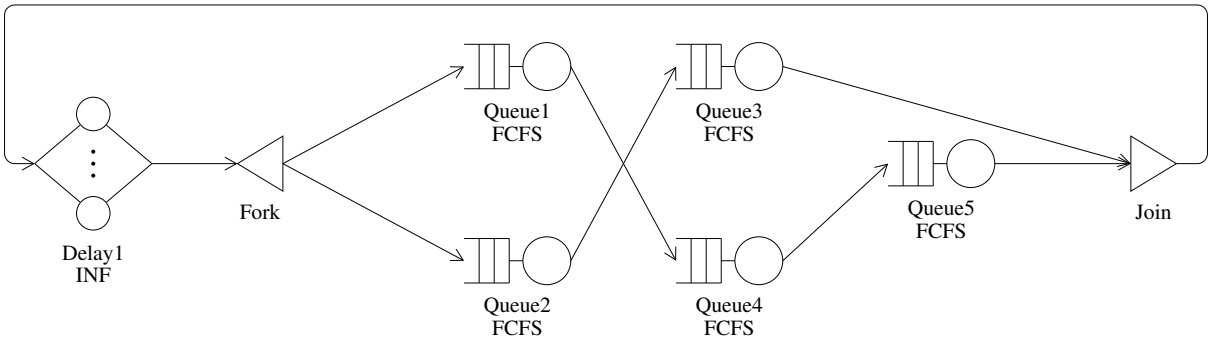

Figure 2.39: Topology of `fj_complex_serial`: a closed network over 6 queueing stations with a fork-join block.

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`.

```
Network model = ForkJoinModel.fj_complex_serial();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 1201.
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

[Running in non-interactive mode, continuing...]
```

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.

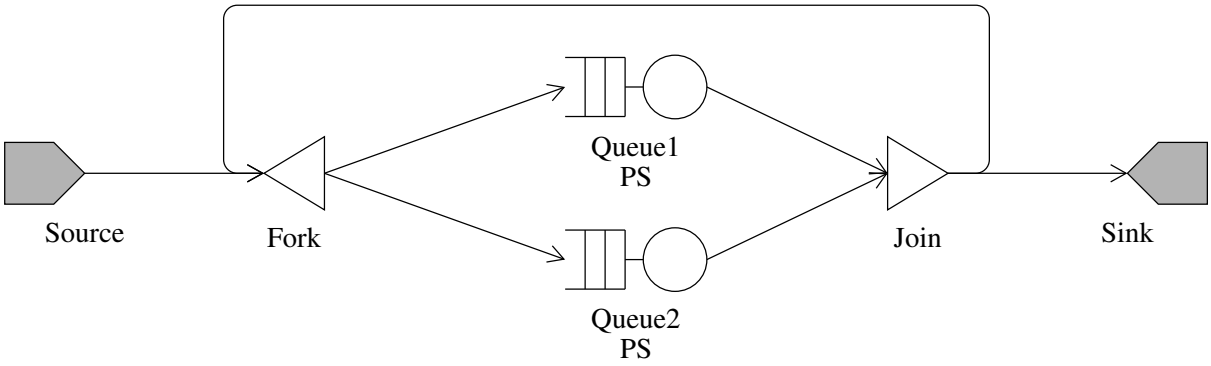

Figure 2.40: Topology of `fj_cs_multi_visits`: an open network over 2 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_cs_multi_visits();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 544.
Station  JobClass     QLen  Util    RespT   ResidT  ArvR  Tput
Source   class1         0     0        0        0     0   0.1
Queue1   class1   0.12474   0.1   1.2474   1.2474   0.1   0.1
Queue1   class2   0.12474   0.1   1.2474   1.2474   0.1   0.1
Queue2   class1   0.12474   0.1   1.2474   1.2474   0.1   0.1
Queue2   class2   0.12474   0.1   1.2474   1.2474   0.1   0.1
Join     class1   0.12474     0  0.62368  0.62368   0.2   0.1
Join     class2   0.12474     0  0.62368  0.62368   0.2   0.1

[Running in non-interactive mode, continuing...]
```

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.

2.6.7 `fj_cs_postfork`

Class switching after the fork, on the sibling tasks.

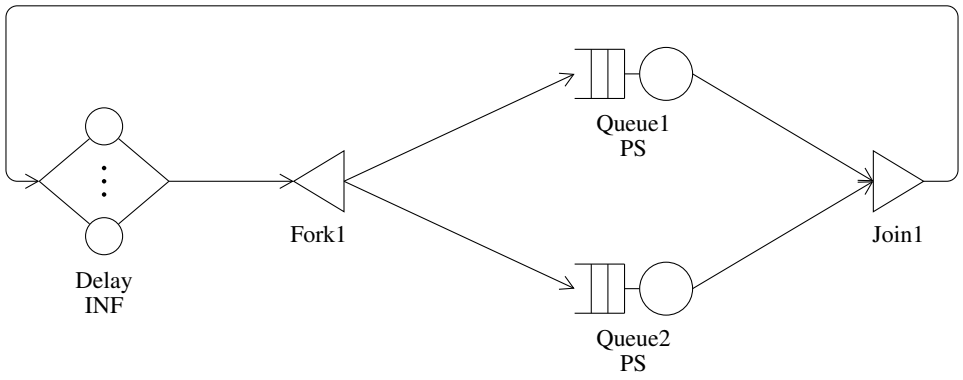

Figure 2.41: Topology of `fj_cs_postfork`: a closed network over 3 queueing stations with a fork-join block.

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`.

The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_cs_postfork();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};
```

```
for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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        0  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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 546.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    class1     1.6364   1.6364        4        4  0.40911  0.40911
Join1    class1    0.24644        0  0.30119  0.30119  0.81822  0.40911
Queue1   class1    0.24644  0.20455  0.60238  0.60238  0.40911  0.40911
Queue2   class1    0.24644  0.20455  0.60238  0.60238  0.40911  0.40911

[Running in non-interactive mode, continuing...]
```

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.

2.6.8 `fj_cs_prefork`

Class switching before the fork.

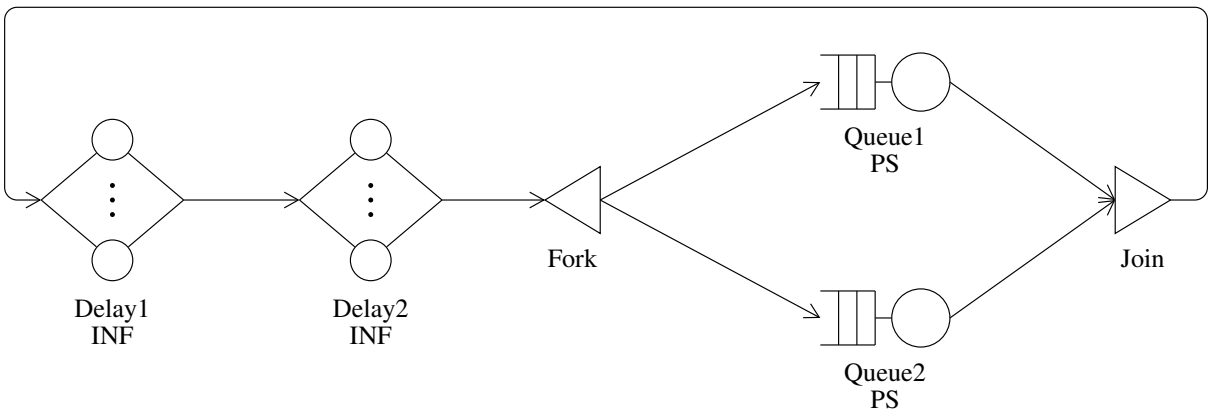

Figure 2.42: Topology of `fj_cs_prefork`: a closed network over 4 queueing stations with a fork-join block.

The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_cs_prefork();
JMT solverJMT = new JMT(model, "seed", 23000);
```

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`.

```
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 887.
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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_deep_nesting();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
```

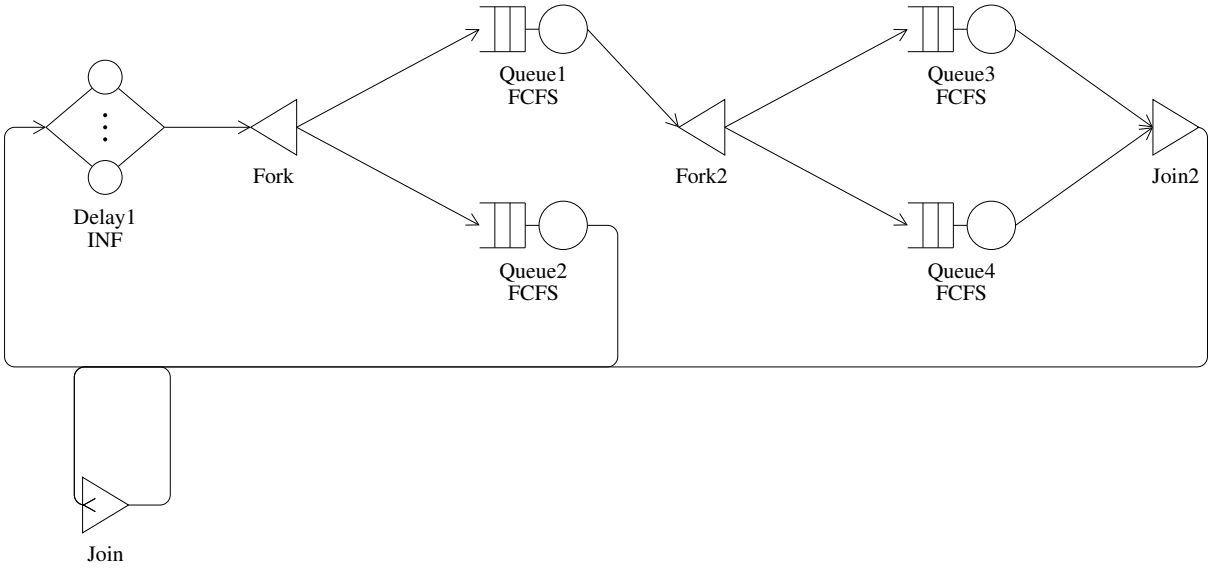

Figure 2.43: Topology of `fj_deep_nesting`: a closed network over 5 queueing stations with a fork-join block.

Nodes	9: Queue $\times$ 4, Delay, Fork $\times$ 2, Join $\times$ 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`.

```
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 549.
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

[Running in non-interactive mode, continuing...]
```

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.

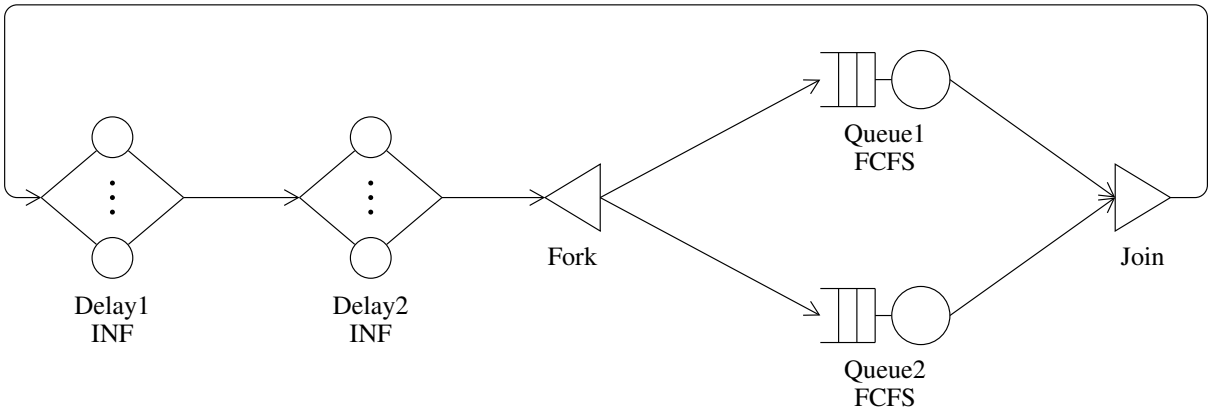

Figure 2.44: Topology of `fj_delays`: a closed network over 4 queueing stations with a fork-join block.

The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_delays();
JMT solverJMT = new JMT(model, "seed", 23000);
```

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`.

```
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 628.
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

[Running in non-interactive mode, continuing...]
```

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.

```
Network model = new Network("ForkJoinOpenClosed");

Source source = new Source(model, "Source");
Delay client = new Delay(model, "Client");
Queue prefork = new Queue(model, "PreFork", SchedStrategy.FCFS);
Queue branch1 = new Queue(model, "Branch1", SchedStrategy.FCFS);
Queue branch2 = new Queue(model, "Branch2", SchedStrategy.FCFS);
Queue postjoin = new Queue(model, "PostJoin", SchedStrategy.FCFS);
Fork fork = new Fork(model, "Fork");
Join join = new Join(model, "Join", fork);
```

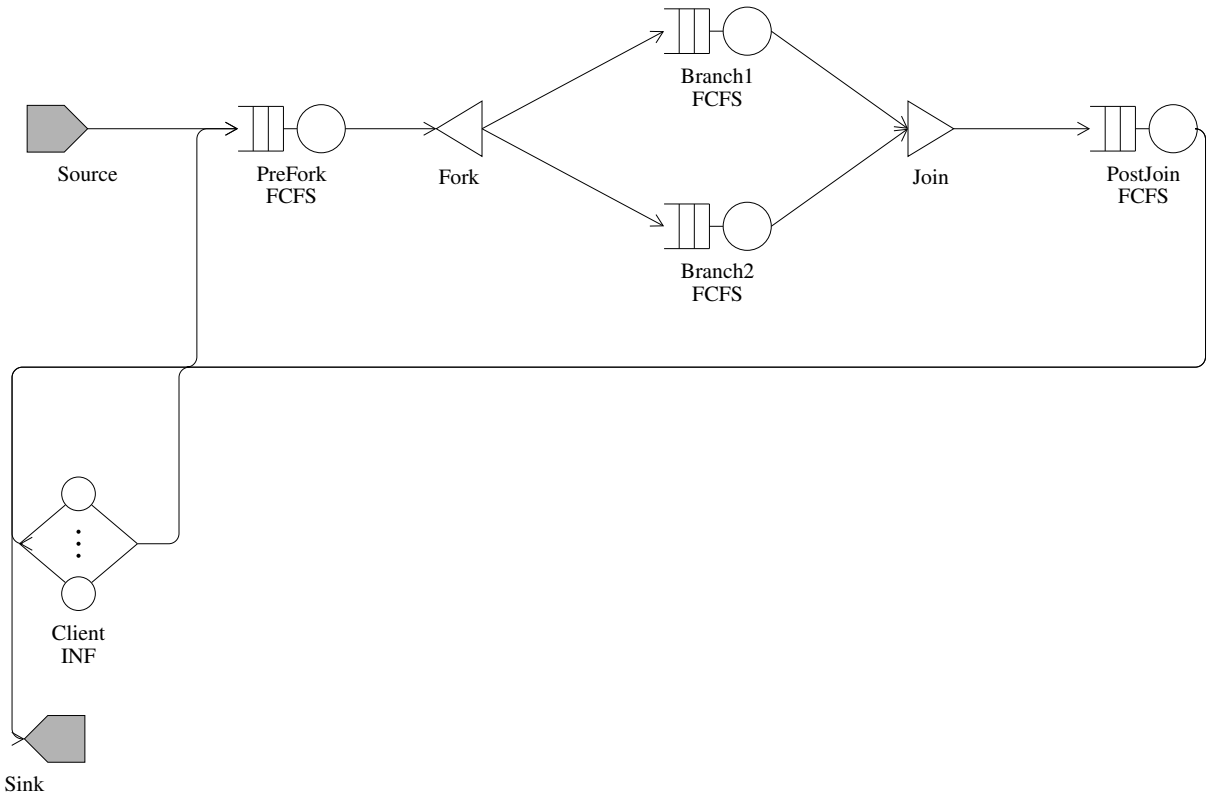

Figure 2.45: Topology of `fj_mixed_openclosed`: an open network over 5 queueing stations with a fork-join block.

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`.

```
Sink sink = new Sink(model, "Sink");

OpenClass oclass = new OpenClass(model, "Open");
ClosedClass cclass = new ClosedClass(model, "Closed", 1, client);

source.setArrival(oclass, new Exp(0.1));
client.setService(cclass, Exp.fitMean(1.0));
client.setService(oclass, Disabled.getInstance());
prefork.setService(oclass, Exp.fitMean(0.2));
prefork.setService(cclass, Exp.fitMean(0.2));
branch1.setService(oclass, Exp.fitMean(0.3));
branch1.setService(cclass, Exp.fitMean(0.3));
branch2.setService(oclass, Exp.fitMean(0.4));
branch2.setService(cclass, Exp.fitMean(0.4));
postjoin.setService(oclass, Exp.fitMean(0.1));
postjoin.setService(cclass, Exp.fitMean(0.1));

    ... (18 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

P.set(cclass, cclass, join, postjoin, 1);
P.set(cclass, cclass, postjoin, client, 1);

model.link(P);

return model;
```

Running this edition prints

```
jline.lang.Network@621be5d1
```

2.6.12 `fj_nojoin`

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

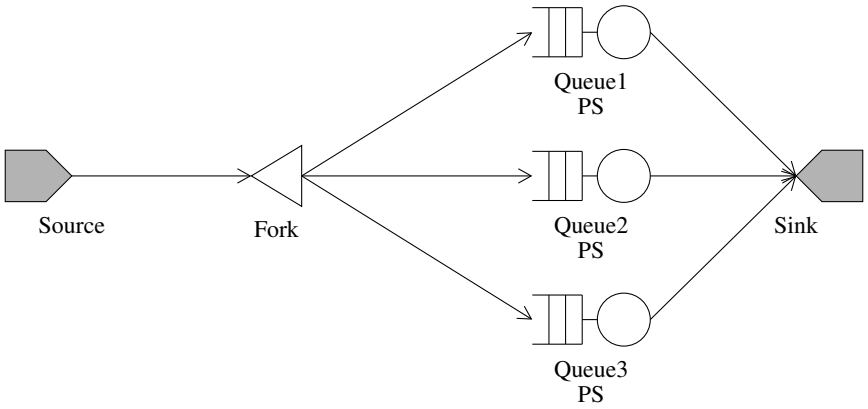

Figure 2.46: Topology of `fj_nojoin`: an open network over 3 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_nojoin();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT Command output: JSIMengine - Warning: A class1 task generated by a fork or scaler reached Sink before
     merging at a join.

JMT analysis [method: default; type: approximate, randomized; lang: java; 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  0.41145  0.41145  0.5062  0.50621
WARNING: [mmt] No join node found for fork node 4.
WARNING: [sort_forks] No join node found for fork node 4 in nested forks processing.
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 1029.
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    0.33314     0.25  0.66629  0.66629   0.5   0.5
Queue3    class1    0.19913  0.16667  0.39826  0.39826   0.5   0.5

[Running in non-interactive mode, continuing...]
```

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.

2.6.13 `fj_route_overlap`

Branches that share a station, so their visits overlap.

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`.

The example is built and solved as follows.

```
Network model = ForkJoinModel.fj_route_overlap();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};
```

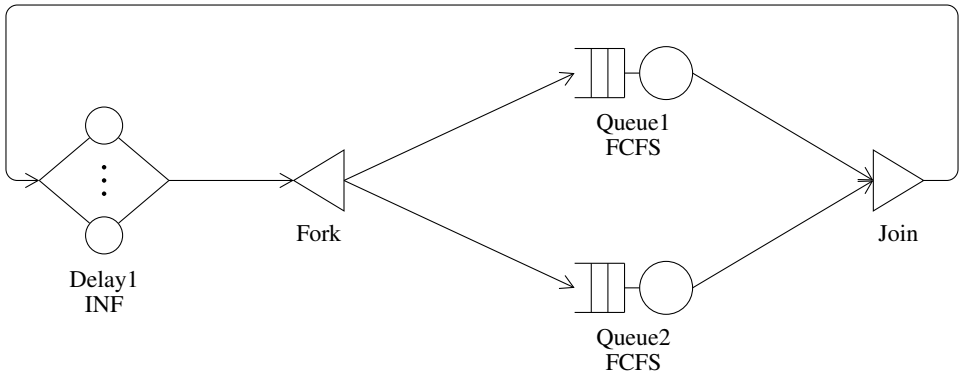

Figure 2.47: Topology of `fj_route_overlap`: a closed network over 3 queueing stations with a fork-join block.

```
for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
     in <t>s. Iterations: 3519.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

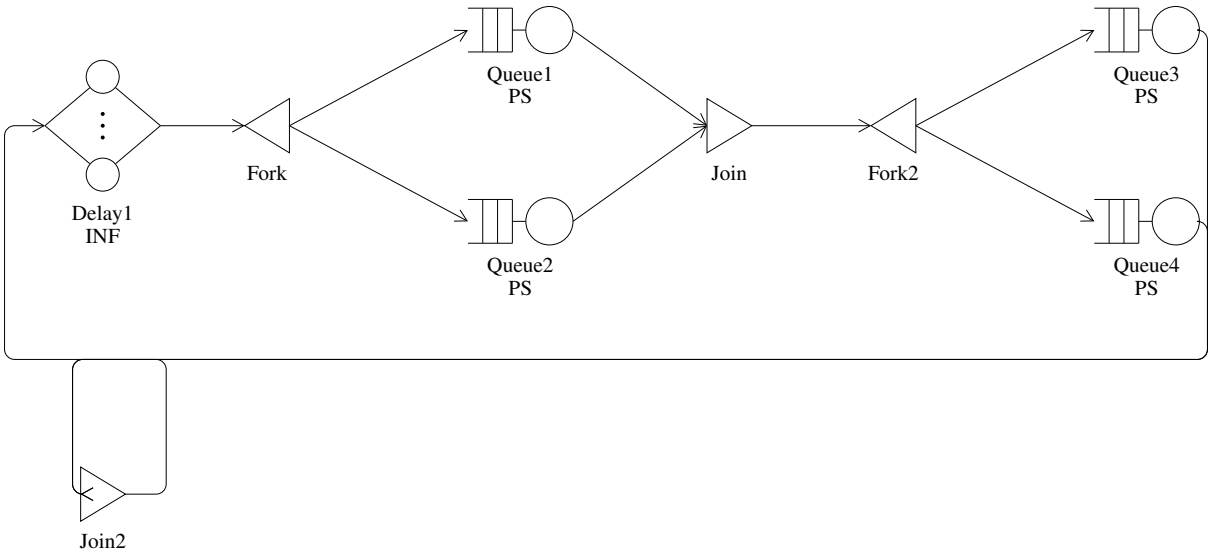

Figure 2.48: Topology of `fj_serialfjs_closed`: a closed network over 5 queueing stations with a fork-join block.

Nodes	9: Queue $\times$ 4, Delay, Fork $\times$ 2, Join $\times$ 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`.

```
Network model = ForkJoinModel.fj_serialfjs_closed();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
     in <t>s. Iterations: 418.
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

[Running in non-interactive mode, continuing...]
```

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.

```
Network model = ForkJoinModel.fj_serialfjs_open();
JMT solverJMT = new JMT(model, "seed", 23000);
```

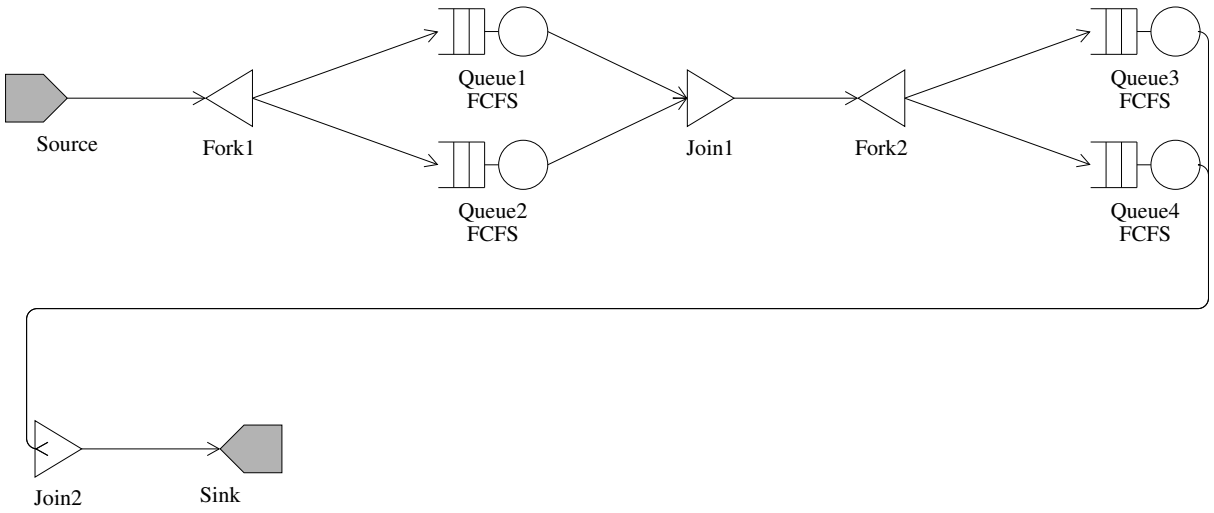

Figure 2.49: Topology of `fj_serialfjs_open`: an open network over 4 queueing stations with a fork-join block.

```
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 809.
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

[Running in non-interactive mode, continuing...]
```

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.

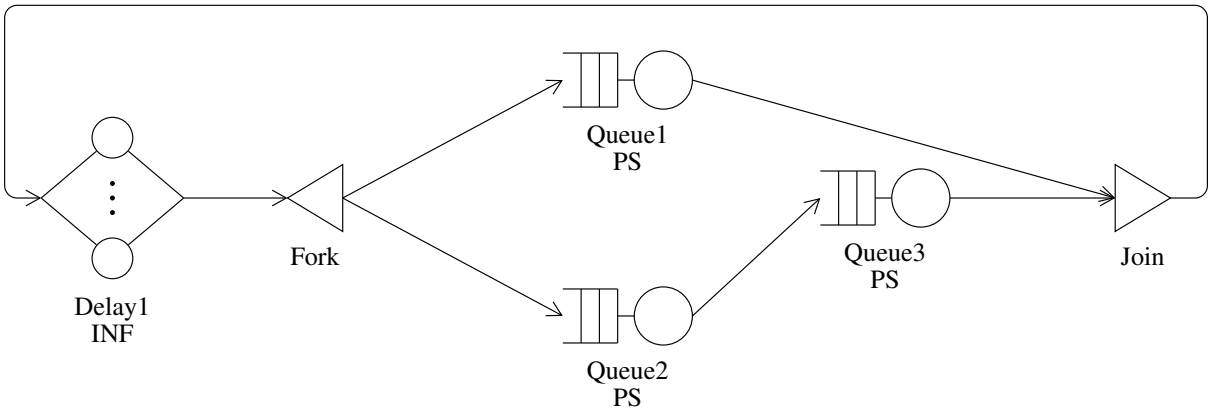

Figure 2.50: Topology of `fj_threebranches`: a closed network over 4 queueing stations with a fork-join block.

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.

```
Network model = ForkJoinModel.fj_threebranches();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT  ResidT     ArvR     Tput
Delay1   class1     1.5598   1.5598   1.9988  1.9988  0.79323  0.79186
Delay1   class2    0.86162  0.86162   1.2766  1.2766  0.67451  0.67426
Queue1   class1     1.8434  0.53176    2.288   2.288  0.79186  0.79224
Queue1   class2    0.94275  0.23676   1.3347  1.3347  0.67426   0.6734
Queue2   class1     4.1168  0.72262   5.2241  5.2241  0.79186  0.79179
```

```
Queue2   class2     1.5395  0.22418  2.2137  2.2137  0.67426  0.67346
Queue3   class1      4.011    0.328   5.052   5.052  0.79179  0.79102
Queue3   class2      7.526  0.67036  11.084  11.084  0.67346  0.67391
Join     class1     6.8098        0  4.1768  4.1768   1.5954  0.79323
Join     class2     8.2743        0  5.9072  5.9072   1.3957  0.67391
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 3224.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay1   class1     1.4591   1.4591       2       2  0.72955  0.72955

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

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

[Running in non-interactive mode, continuing...]
```

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.

2.6.17 `fj_tiny_closed`

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

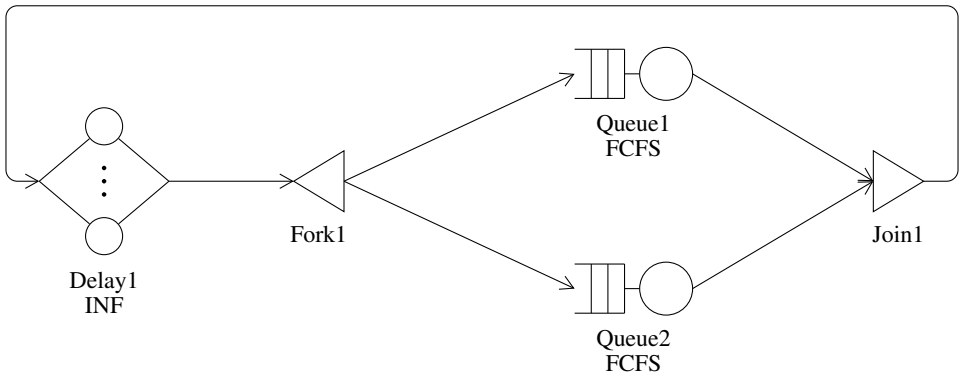

Figure 2.51: Topology of `fj_tiny_closed`: a closed network over 3 queueing stations with a fork-join block.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.6.18 `fj_twoclasses_forked`

Two classes forked independently through the same block.
The example is built and solved as follows.

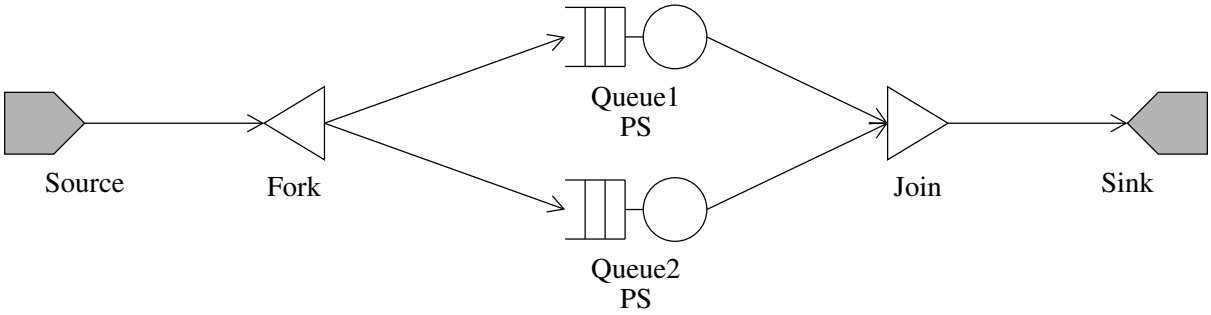

Figure 2.52: Topology of `fj_twoclasses_forked`: an open network over 2 queueing stations with a fork-join block.

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`.

```
Network model = ForkJoinModel.fj_twoclasses_forked();
JMT solverJMT = new JMT(model, "seed", 23000);
MVA solverMVA = new MVA(model);

Object[] solvers = {solverJMT, solverMVA};

for (Object solverObj : solvers) {
    try {
        if (solverObj instanceof JMT) {
            JMT solver = (JMT) solverObj;
            solver.getAvgTable().print();
        } else if (solverObj instanceof MVA) {
            MVA solver = (MVA) solverObj;
            solver.getAvgTable().print();
        }
    } catch (Exception e) {
        System.out.println("Error with solver: " + e.getMessage());
    }
}
pauseForUser();
```

Running this edition prints

```
WARNING: [setTasksPerLink] The setTasksPerLink feature is experimental and results may be inaccurate for
     analytical solvers.
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 7220.
Station  JobClass    QLen     Util    RespT  ResidT  ArvR  Tput
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   2e-08       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    15.082        0  15.082  15.082     1  0.25
Join     class2         6        0       6       6     1  0.25

[Running in non-interactive mode, continuing...]
```

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.

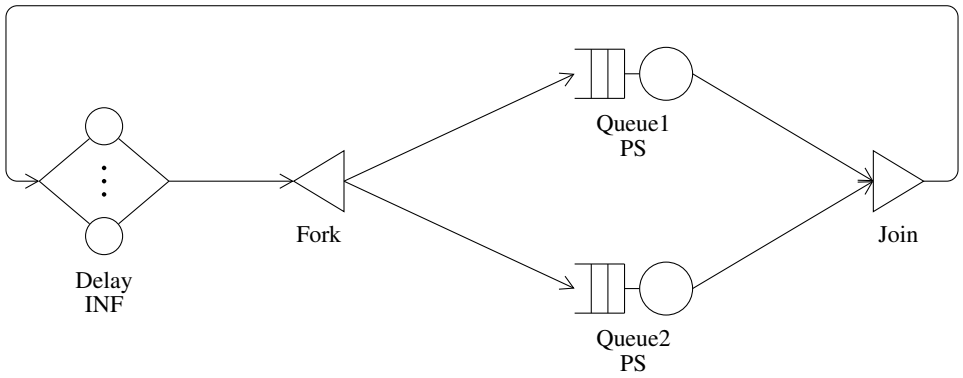

Figure 2.53: Topology of `fj_variable_fanout`: a closed network over 3 queueing stations with a fork-join block.

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.

```
Network model = new Network("model");
Delay delay = new Delay(model, "Delay");
Queue queue1 = new Queue(model, "Queue1", SchedStrategy.PS);
Queue queue2 = new Queue(model, "Queue2", SchedStrategy.PS);
Fork fork = new Fork(model, "Fork");
Join join = new Join(model, "Join", fork);

ClosedClass jobclass1 = new ClosedClass(model, "class1", 4, delay);

delay.setService(jobclass1, new Exp(1.0));
queue1.setService(jobclass1, new Exp(2.0));
queue2.setService(jobclass1, new Exp(2.0));

if ("vector".equals(mode)) {
    fork.setTasksPerLink(jobclass1, 3.0);
} else if ("random".equals(mode)) {
    Matrix pmf = new Matrix(1, 2);
    pmf.set(0, 0, 0.5);
    pmf.set(0, 1, 0.5);
    Matrix pts = new Matrix(1, 2);
```

```
    pts.set(0, 0, 1.0);
    pts.set(0, 1, 3.0);
    fork.setTasksPerLinkDistribution(jobclass1, new DiscreteSampler(pmf, pts));
} else if ("prob".equals(mode)) {
    // an uncertain branch needs a Join that does not wait for it
    join.setStrategy(jobclass1, JoinStrategy.PARTIAL);

    ... (8 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

routingMatrix.set(jobclass1, jobclass1, queue1, join, 1.0);
routingMatrix.set(jobclass1, jobclass1, queue2, join, 1.0);
routingMatrix.set(jobclass1, jobclass1, join, delay, 1.0);
model.link(routingMatrix);

return model;
```

Running this edition prints

```
no no-arg static method fj_variable_fanout on jline.examples.java.basic.ForkJoinModel
```

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.

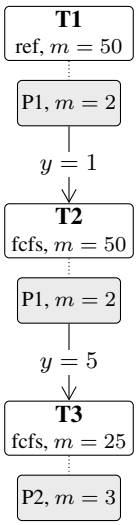

Figure 2.54: Call graph of `lqn_basic`: 3 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

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`.

```
public static void lqn_basic() throws Exception {
    LayeredNetwork model = LayeredModel.lqn_basic();

    // The reference solves this with the layered solver on its default layers.
    new LN(model).getAvgTable().print();

    pauseForUser();
```

Running this edition prints

```
Iter  1.  Analyze time: 0.302s. Update time: 0.021s. Runtime: 0.304s.
Iter  2.  Analyze time: 0.186s. Update time: 0.006s. Runtime: 0.516s.
Iter  3.  Analyze time: 0.174s. Update time: 0.017s. Runtime: 0.696s.
Iter  4.  Analyze time: 0.142s. Update time: 0.011s. Runtime: 0.860s.
Iter  5.  Analyze time: 0.173s. Update time: 0.022s. Runtime: 1.058s.
Iter  6.  Analyze time: 0.158s. Update time: 0.028s. Runtime: 1.245s.
Iter  7.  Analyze time: 0.138s. Update time: 0.004s. Runtime: 1.412s.
Iter  8.  Analyze time: 0.129s. Update time: 0.011s. Runtime: 1.549s.
Iter  9.  Analyze time: 0.073s. Update time: 0.007s. Runtime: 1.634s.
Iter 10.  Analyze time: 0.127s. Update time: 0.005s. Runtime: 1.773s.
Iter 11.  Analyze time: 0.117s. Update time: 0.017s. Runtime: 1.897s.
Iter 12.  Analyze time: 0.180s. Update time: 0.014s. Runtime: 2.099s.
Iter 13.  Analyze time: 0.136s. Update time: 0.006s. Runtime: 2.250s.
Iter 14.  Analyze time: 0.216s. Update time: 0.012s. Runtime: 2.474s.
Iter 15.  Analyze time: 0.149s. Update time: 0.007s. Runtime: 2.636s.
Iter 16.  Analyze time: 0.117s. Update time: 0.003s. Runtime: 2.770s.

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

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

[Running in non-interactive mode, continuing...]
```

2.7.2 `lqn_bpmn`

A BPMN process translated into a layered model with forks, joins and immediate activities.
The example is built and solved as follows.

```
LayeredNetwork model = LayeredModel.lqn_bpmn();

LQNS solver = new LQNS(model);
AvgTable avgTable = solver.getAvgTable();
avgTable.print();

pauseForUser();
```

Running this edition prints

```
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
```

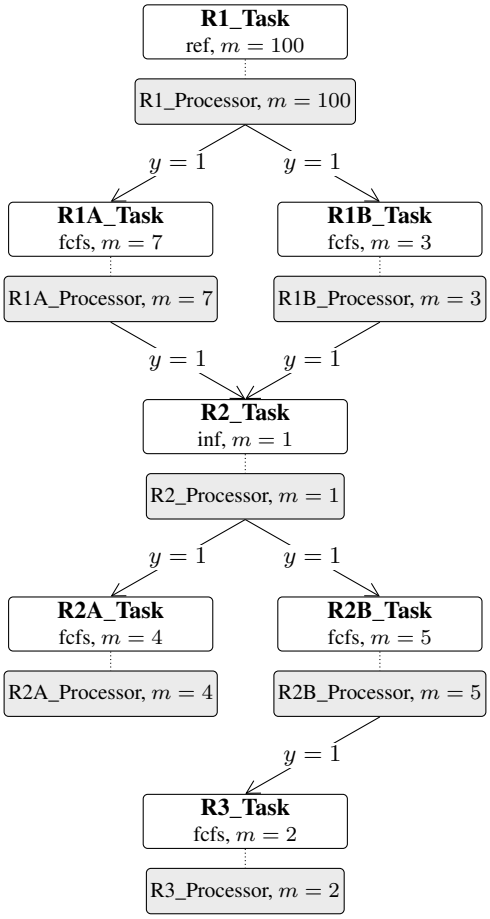

Figure 2.55: Call graph of `lqn_bpmn`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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`.

```
R2A_Processor        Processor      NaN   1.3324     NaN     NaN   NaN       NaN
R2B_Processor        Processor      NaN   1.7575     NaN     NaN   NaN       NaN
R1_Task              RefTask      98.06        0     NaN     NaN   NaN   0.14365
R2_Task              Task         6.173        0     NaN     NaN   NaN   0.27712
R3_Task              Task        1.2198   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
R2A_Task             Task        1.3332   1.3324     NaN     NaN   NaN   0.21794
R2B_Task             Task        3.0992   1.7575     NaN     NaN   NaN   0.31185
R1_Ref_Entry         Entry        98.06        0  682.64     NaN   NaN   0.14365

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

A8                   Activity   0.64744  0.64716  8.0035     NaN   NaN  0.080894
A11                  Activity   0.27434   0.2741  4.0035     NaN   NaN  0.068525
A9                   Activity   0.49165  0.49153  4.0009     NaN   NaN   0.12288
A10                  Activity   0.73741   0.7373  6.0009     NaN   NaN   0.12288
A12                  Activity    0.5287  0.52864  8.0009     NaN   NaN   0.06608
A9_Res_Empty         Activity    1.3414        0  10.916     NaN   NaN   0.12288

[Running in non-interactive mode, continuing...]
```

2.7.3 `lqn_bpmn_trace`

The BPMN translation with the activity trace printed.

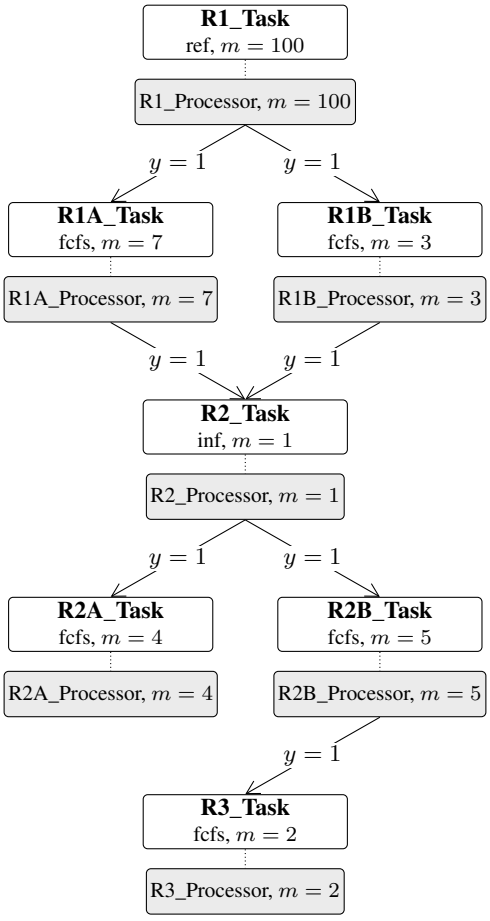

Figure 2.56: Call graph of `lqn_bpmn_trace`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

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`.

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

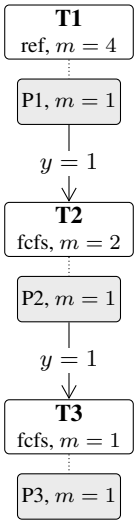

Figure 2.57: Call graph of `lqn_flatph`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.7.5 `lqn_fork_open_arrival`

An AND-fork under an open arrival stream.
The example is built and solved as follows.

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`.

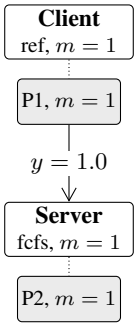

Figure 2.58: Call graph of `lqn_fork_open_arrival`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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`.

```
LayeredNetwork model = new LayeredNetwork("lqnForkOpenArrival");

Processor P1 = new Processor(model, "P1", Integer.MAX_VALUE, SchedStrategy.INF);
Processor P2 = new Processor(model, "P2", Integer.MAX_VALUE, SchedStrategy.INF);

Task T1 = new Task(model, "Client", 1, SchedStrategy.REF).on(P1);
T1.setThinkTime(Exp.fitMean(1.0));
Task T2 = new Task(model, "Server", 1, SchedStrategy.FCFS).on(P2);
T2.setThinkTime(Immediate.getInstance());

Entry CE = new Entry(model, "CE").on(T1);
Entry SE = new Entry(model, "SE").on(T2);
Entry OE = new Entry(model, "OE").on(T2);
OE.setArrival(new Exp(0.1));

new Activity(model, "CA", Exp.fitMean(0.5)).on(T1).boundTo(CE).synchCall(SE, 1);

new Activity(model, "RA1", Exp.fitMean(0.2)).on(T2).boundTo(SE);
new Activity(model, "RA2", Exp.fitMean(0.3)).on(T2);
new Activity(model, "RA3", Exp.fitMean(0.4)).on(T2);
new Activity(model, "RA4", Exp.fitMean(0.1)).on(T2).repliesTo(SE);

new Activity(model, "OA1", Exp.fitMean(0.2)).on(T2).boundTo(OE);
new Activity(model, "OA2", Exp.fitMean(0.3)).on(T2);
new Activity(model, "OA3", Exp.fitMean(0.4)).on(T2);
new Activity(model, "OA4", Exp.fitMean(0.1)).on(T2);

    ... (8 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

openBranches.add("OA2");
openBranches.add("OA3");
T2.addPrecedence(ActivityPrecedence.AndFork("OA1", openBranches));
T2.addPrecedence(ActivityPrecedence.AndJoin(openBranches, "OA4"));

return model;
```

Running this edition prints

```
jline.lang.layered.LayeredNetwork@533ddba
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.7.7 `lqn_moment3`

The third moment of the entry service time, and how it propagates up the layers.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

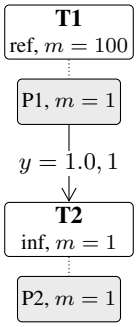

Figure 2.59: Call graph of `lqn_moment3`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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`.

2.7.8 `lqn_multi_solvers`

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

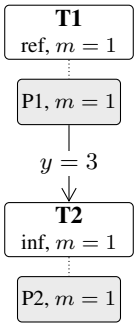

Figure 2.60: Call graph of `lqn_multi_solvers`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
LayeredNetwork model = LayeredModel.lqn_multi_solvers();

// LQNS solver as in MATLAB (lines 27-29)
SolverOptions lqnsOptions = LQNS.defaultOptions();
lqnsOptions.keep = true;
lqnsOptions.verbose = VerboseLevel.STD;
LQNS lqnsSolver = new LQNS(model, lqnsOptions);
AvgTable avgTableLQNS = lqnsSolver.getAvgTable();
avgTableLQNS.print();

// LN with MVA solver in each layer (lines 37-39)
SolverOptions lnOptions = LN.defaultOptions();
lnOptions.verbose = VerboseLevel.SILENT;
lnOptions.seed = 2300;
SolverOptions mvaOptions = MVA.defaultOptions();
```

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`.

```
mvaOptions.verbose = VerboseLevel.SILENT;
LN solverLN_MVA = new LN(model, (subModel) -> new MVA(subModel, mvaOptions), lnOptions);
AvgTable avgTableLN_MVA = solverLN_MVA.getAvgTable();
avgTableLN_MVA.print();

// LN with NC solver in each layer (lines 47-49)
SolverOptions ncOptions = NC.defaultOptions();
ncOptions.verbose = VerboseLevel.SILENT;
LN solverLN_NC = new LN(model, (subModel) -> new NC(subModel, ncOptions), lnOptions);
AvgTable avgTableLN_NC = solverLN_NC.getAvgTable();
avgTableLN_NC.print();

pauseForUser();
```

Running this edition prints

```
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    0.99998  0.24999    NaN     NaN   NaN  0.24999
T2    Task       0.74998  0.74998    NaN     NaN   NaN  0.74998
E1    Entry      0.99998  0.24999      4     NaN   NaN  0.24999
E2    Entry      0.74998  0.74998      1     NaN   NaN  0.74998
A1    Activity   0.99998  0.24999      4     NaN   NaN  0.24999
A2    Activity   0.74998  0.74998      1     NaN   NaN  0.74998

[Running in non-interactive mode, continuing...]
```

2.7.9 `lqn_ofbiz`

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

```
LayeredNetwork model = LayeredModel.lqn_ofbiz();

try {
    new LQNS(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LQNS failed: " + e.getMessage());
}
// NC layers: the golden keys this row LN(NC).
new LN(model, (subModel) -> new NC(subModel, "verbose", false)).getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
SEVERE: [parseXML] File cannot be found. Verify the current directory and the specified filename.
LQNS failed: Cannot invoke "jline.lang.layered.LayeredNetwork.writeXML(String, boolean)" because "this.
    model" is null
FAILED lqn_ofbiz: java.lang.NullPointerException: Cannot invoke "jline.lang.layered.LayeredNetwork.
    getStruct()" because "lqnmodel" is null
```

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`.

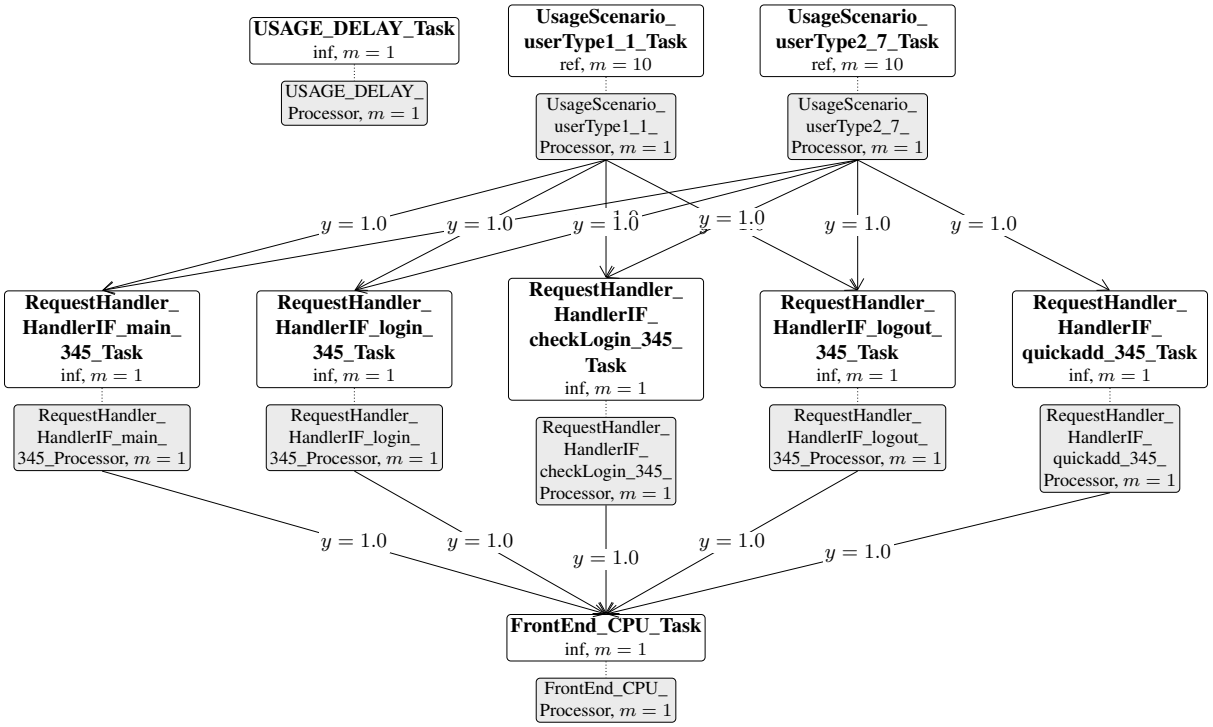

Figure 2.61: Call graph of `lqn_ofbiz`: 9 tasks over 9 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

2.7.10 `lqn_open_arrival`

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

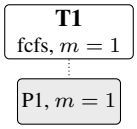

Figure 2.62: Call graph of `lqn_open_arrival`: 1 task over 1 processor, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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.

```
LayeredNetwork model = new LayeredNetwork("openArrivalLQN");

Processor P1 = new Processor(model, "P1", 1, SchedStrategy.PS);
Task T1 = new Task(model, "T1", 1, SchedStrategy.FCFS).on(P1);
T1.setThinkTime(Immediate.getInstance());
Entry E1 = new Entry(model, "E1").on(T1);
E1.setArrival(new Exp(0.2));

new Activity(model, "A1", Exp.fitMean(1.6)).on(T1).boundTo(E1).repliesTo(E1);
```

```
return model;
```

Running this edition prints

```
jline.lang.layered.LayeredNetwork@377dca04
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.7.13 lqn_serial

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

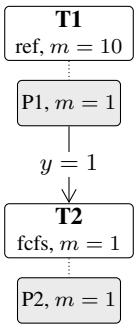

Figure 2.63: Call graph of `lqn_serial`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
LayeredNetwork model = LayeredModel.lqn_serial();

// Create LQNS solver as in MATLAB (lines 32-34)
SolverOptions options = LQNS.defaultOptions();
options.keep = true;
LQNS solver = new LQNS(model, options);
```

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`.

```
AvgTable avgTable = solver.getAvgTable();
avgTable.print();

// Get raw avg tables as in MATLAB (lines 37-39)
AvgTable[] rawTables = solver.getRawAvgTables();
rawTables[0].print();
if (rawTables.length > 1) {
    rawTables[1].print();
}

// LN as in MATLAB (line 41)
LN solverLN = new LN(model, SolverType.MVA);
AvgTable avgTableLN = solverLN.getAvgTable();
avgTableLN.print();

pauseForUser();
```

Running this edition prints

```
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    0.16272   0.14206  1.8327     NaN   NaN  0.088786
AS2   Activity    0.95874         0  10.798     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
Node  NodeType       Util   P1Util  P2Util  P1SvcT  P2SvcT      Tput  ProcWait  ProcUtil
P1    Processor         0        0       0       0       0         0         0   0.14206
P2    Processor         0        0       0       0       0         0         0   0.53271
T1    Task         1.1215   1.1215       0       0       0  0.088786         0   0.14206
T2    Task        0.53271  0.53271       0       0       0  0.088786         0   0.53271

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

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

[Running in non-interactive mode, continuing...]
```

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.

```
SolverOptions opt = LN.defaultOptions();
```

```
// a compatibility declaration is a station rate law, which only the
// class-switching layerings can carry
opt.method = "srvn.cs";

System.out.println("--- homogeneous pool on P1 ---");
new LN(LayeredModel.lqn_server_pools(false), opt).getAvgTable().print();

System.out.println("--- compatibility pool on P1 ---");
new LN(LayeredModel.lqn_server_pools(true), opt).getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
--- homogeneous pool on P1 ---

Iter  1.  Analyze time: 0.271s. Update time: 0.027s. Runtime: 0.272s.
Iter  2.  Analyze time: 0.167s. Update time: 0.023s. Runtime: 0.467s.
Iter  3.  Analyze time: 0.140s. Update time: 0.029s. Runtime: 0.631s.
Iter  4.  Analyze time: 0.069s. Update time: 0.011s. Runtime: 0.735s.
Iter  5.  Analyze time: 0.036s. Update time: 0.026s. Runtime: 0.783s.
Iter  6.  Analyze time: 0.043s. Update time: 0.022s. Runtime: 0.852s.
Iter  7.  Analyze time: 0.069s. Update time: 0.022s. Runtime: 0.944s.
Iter  8.  Analyze time: 0.083s. Update time: 0.028s. Runtime: 1.055s.
Iter  9.  Analyze time: 0.071s. Update time: 0.018s. Runtime: 1.159s.
Iter 10.  Analyze time: 0.081s. Update time: 0.032s. Runtime: 1.270s.
Iter 11.  Analyze time: 0.064s. Update time: 0.029s. Runtime: 1.366s.
Iter 12.  Analyze time: 0.069s. Update time: 0.028s. Runtime: 1.467s.
Iter 13.  Analyze time: 0.096s. Update time: 0.031s. Runtime: 1.595s.
Iter 14.  Analyze time: 0.078s. Update time: 0.023s. Runtime: 1.722s.

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

E1    Entry       1.7208  0.63961   1.3452      NaN   NaN  1.2792
E2    Entry      0.28853  0.14212  0.22558      NaN   NaN   1.279
E3    Entry       0.4248  0.21311  0.33222      NaN   NaN  1.2787
A1    Activity    1.7225  0.63961   1.3465  0.78479   NaN  1.2792
A2    Activity   0.28853  0.14212  0.22558  0.22558   NaN   1.279
A3    Activity    0.4248  0.21311  0.33222  0.33222   NaN  1.2787

[Running in non-interactive mode, continuing...]
```

2.7.15 `lqn_setup`

Task setup and initialization activities before service.

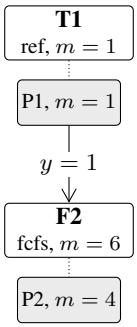

Figure 2.64: Call graph of `lqn_setup`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
LayeredNetwork model = LayeredModel.lqn_setup();

// The reference drives MVA layers, which is what its golden holds.
new LN(model, SolverType.MVA).getAvgTable().print();
```

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`.

```
pauseForUser();
```

Running this edition prints

```
pfqn_mvald: MVA-LD is numerically unstable on this model, LINE will force all probabilities to be non-
    negative.

Iter  1.  Analyze time: 0.110s. Update time: 0.017s. Runtime: 0.111s.
Iter  2.  Analyze time: 0.023s. Update time: 0.018s. Runtime: 0.155s.
Iter  3.  Analyze time: 0.025s. Update time: 0.013s. Runtime: 0.199s.
Iter  4.  Analyze time: 0.026s. Update time: 0.025s. Runtime: 0.239s.
Iter  5.  Analyze time: 0.019s. Update time: 0.010s. Runtime: 0.288s.
Iter  6.  Analyze time: 0.018s. Update time: 0.013s. Runtime: 0.321s.
Iter  7.  Analyze time: 0.025s. Update time: 0.014s. Runtime: 0.372s.
Iter  8.  Analyze time: 0.034s. Update time: 0.015s. Runtime: 0.422s.
Iter  9.  Analyze time: 0.012s. Update time: 0.008s. Runtime: 0.451s.
Iter 10.  Analyze time: 0.007s. Update time: 0.029s. Runtime: 0.471s.
Iter 11.  Analyze time: 0.016s. Update time: 0.012s. Runtime: 0.523s.
Iter 12.  Analyze time: 0.013s. Update time: 0.024s. Runtime: 0.550s.
Iter 13.  Analyze time: 0.016s. Update time: 0.020s. Runtime: 0.594s.
Iter 14.  Analyze time: 0.009s. Update time: 0.011s. Runtime: 0.625s.

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

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

[Running in non-interactive mode, continuing...]
```

2.7.16 `lqn_sockshop`

The Sock Shop microservice benchmark as a layered model.

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`.

The example is built and solved as follows.

```
LayeredNetwork model = LayeredModel.lqn_sockshop();

LN solverLN = new LN(model, SolverType.MVA);
AvgTable avgTable = solverLN.getAvgTable();
avgTable.print();

pauseForUser();
```

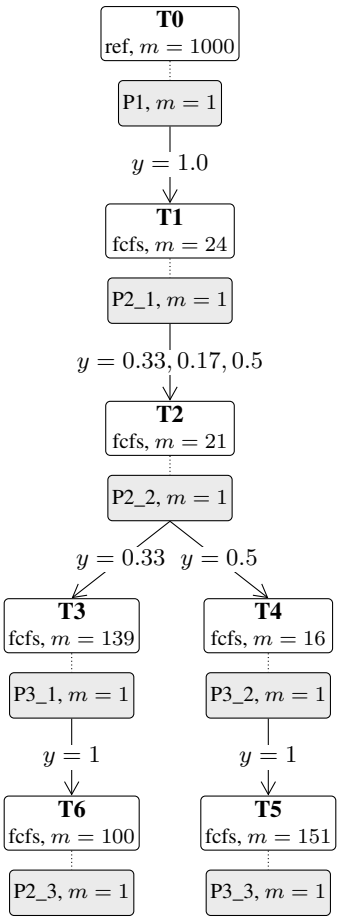

Figure 2.65: Call graph of `lqn_sockshop`: 7 tasks over 7 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Running this edition prints

```
Iter  1.  Analyze time: 0.262s. Update time: 0.113s. Runtime: 0.271s.
Iter  2.  Analyze time: 0.154s. Update time: 0.067s. Runtime: 0.542s.
Iter  3.  Analyze time: 0.137s. Update time: 0.074s. Runtime: 0.755s.
Iter  4.  Analyze time: 0.268s. Update time: 0.074s. Runtime: 1.104s.
Iter  5.  Analyze time: 0.323s. Update time: 0.039s. Runtime: 1.506s.
Iter  6.  Analyze time: 0.149s. Update time: 0.069s. Runtime: 1.701s.
Iter  7.  Analyze time: 0.253s. Update time: 0.072s. Runtime: 2.035s.
Iter  8.  Analyze time: 0.280s. Update time: 0.042s. Runtime: 2.392s.
Iter  9.  Analyze time: 0.063s. Update time: 0.030s. Runtime: 2.502s.
Iter 10.  Analyze time: 0.080s. Update time: 0.042s. Runtime: 2.614s.
Iter 11.  Analyze time: 0.082s. Update time: 0.030s. Runtime: 2.744s.
Iter 12.  Analyze time: 0.080s. Update time: 0.023s. Runtime: 2.857s.
Iter 13.  Analyze time: 0.075s. Update time: 0.027s. Runtime: 2.962s.
Iter 14.  Analyze time: 0.050s. Update time: 0.030s. Runtime: 3.042s.
Iter 15.  Analyze time: 0.043s. Update time: 0.029s. Runtime: 3.119s.
Iter 16.  Analyze time: 0.060s. Update time: 0.046s. Runtime: 3.209s.

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

AS3   Activity   1.3068e-07  1.2107e-07  1.0793e-08           0   NaN   12.1
AS4   Activity     0.088229    0.042285   0.0072872   0.0018848   NaN   12.1
AS5   Activity   1.3068e-07  1.2107e-07  1.0793e-08           0   NaN   12.1
AS6   Activity     0.082004    0.036518   0.0067731   0.0016277   NaN   12.1
AH13  Activity   2.6262e-07  2.4214e-07  1.0846e-08  1.0846e-08   NaN   24.2
AH14  Activity     0.085177    0.078536   0.0035177   0.0035177   NaN   24.2

[Running in non-interactive mode, continuing...]
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.7.18 `lqn_transient`

The transient response of a layered model, through `getTranAvg`.

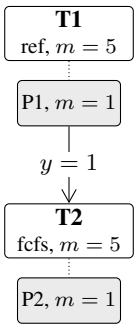

Figure 2.66: Call graph of `lqn_transient`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

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`.

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.7.19 `lqn_twotasks`

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

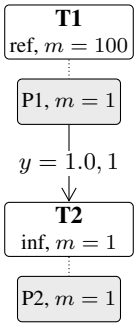

Figure 2.67: Call graph of `lqn_twotasks`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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.

```
LayeredNetwork model = LayeredModel.lqn_twotasks();

try {
    new LQNS(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LQNS failed: " + e.getMessage());
}
// NC LAYERS, NOT THE DEFAULT: MVA layers and NC layers are different fixed
// points, so the layer solver the reference pins is part of the golden.
new LN(model, (subModel) -> new NC(subModel, "verbose", false)).getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
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       97.168     1     NaN     NaN   NaN   0.5
E1    Entry        97.5  0.25     390     NaN   NaN  0.25
E2    Entry      72.876  0.75   291.5     NaN   NaN  0.25
E3    Entry      24.292  0.25  97.168     NaN   NaN  0.25
A1    Activity     97.5  0.25     390     NaN   NaN  0.25
A20   Activity   24.292  0.25  97.168     NaN   NaN  0.25
A21   Activity   24.292  0.25  97.168     NaN   NaN  0.25
A22   Activity   24.292  0.25  97.168     NaN   NaN  0.25
A3    Activity   24.292  0.25  97.168     NaN   NaN  0.25

Iter  1.  Analyze time: 0.399s. Update time: 0.019s. Runtime: 0.400s.
Iter  2.  Analyze time: 0.124s. Update time: 0.020s. Runtime: 0.552s.

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

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

[Running in non-interactive mode, continuing...]
```

2.7.20 `lqn_workflows`

Workflow patterns (sequence, AND-fork and join, loop) expressed as activity graphs.

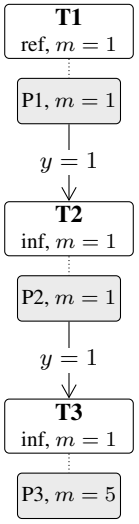

Figure 2.68: Call graph of `lqn_workflows`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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.

```
LayeredNetwork model = LayeredModel.lqn_workflows();
```

```
try {
    new LQNS(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("LQNS failed: " + e.getMessage());
}
new LN(model).getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
WARNING: [writeXML] Task T3 is not a reference task, so its think time (10.0000) is not written: the LQN
     XML schema accepts think-time on reference tasks only.
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
A3     Activity      0.44297    0.23873     5.5667     NaN   NaN   0.079575
B1     Activity    0.0079575  0.0079575        0.1     NaN   NaN   0.079575
B2     Activity     0.015915   0.015915        0.2     NaN   NaN   0.079575

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

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

[Running in non-interactive mode, continuing...]
```

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.

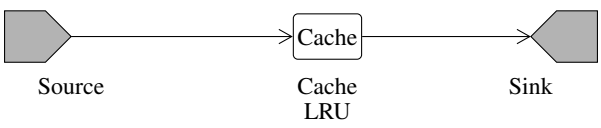

Figure 2.69: Topology of `cache_compare_replc`: an open network with a cache node.

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.

```
public static void cache_compare_replc() {
    System.out.println("=== Cache Compare Replacement Strategies ===");

    ReplacementStrategy[] replStrat = {ReplacementStrategy.RR, ReplacementStrategy.FIFO,
    ReplacementStrategy.LRU};

    for (int s = 0; s < replStrat.length; s++) {

        try {
            // Create a fresh model for each strategy (replacement strategy is set in
    constructor)
            Network model;
            if (replStrat[s] == ReplacementStrategy.RR) {
                model = CacheModel.cache_replc_rr();
            } else if (replStrat[s] == ReplacementStrategy.FIFO) {
                model = CacheModel.cache_replc_fifo();
            } else if (replStrat[s] == ReplacementStrategy.LRU) {
                model = CacheModel.cache_replc_lru();
            } else {
                model = CacheModel.cache_compare_replc();
            }
            jline.lang.nodes.Cache cacheNode = (jline.lang.nodes.Cache) model.getNodeByName("
    Cache");

            // Skip CTMC solver as commented in the source notebook

            model.reset();
            MVA solver2 = new MVA(model, "seed", 1, "verbose", false);
            NetworkAvgNodeTable avgTable2 = solver2.getAvgNodeTable();

    ... (12 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

            System.out.printf("%s: MVA=%.8f, NC=%.8f%n", replStrat[s], mvaHitRatio,
    ncHitRatio);

        } catch (Exception e) {
            System.out.println("Error testing " + replStrat[s] + ": " + e.getMessage());
        }
    }
```

The captured run completed successfully and produced no console output.

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.

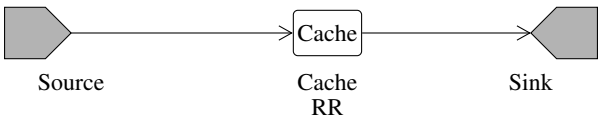

Figure 2.70: Topology of `cache_itemsize_costcap`: an open network with a cache node.

The example is built and solved as follows.

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`.

```
Network model = new Network("model");

int n = 6;                    // number of items
Matrix m = new Matrix(1, 2); // two lists, one item each
m.set(0, 0, 1);
m.set(0, 1, 1);

Matrix sizes = new Matrix(1, n); // small and large items
double[] sizeValues = {1, 1, 1, 2, 2, 2};
for (int i = 0; i < n; i++) sizes.set(0, i, sizeValues[i]);
Matrix caps = new Matrix(1, 2); // list 2 admits small items only
caps.set(0, 0, 2);
caps.set(0, 1, 1);

Source source = new Source(model, "Source");
Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.RR);
Sink sink = new Sink(model, "Sink");

OpenClass jobClass = new OpenClass(model, "InitClass", 0);
OpenClass hitClass = new OpenClass(model, "HitClass", 0);
OpenClass missClass = new OpenClass(model, "MissClass", 0);

source.setArrival(jobClass, new Exp(2.0));
cacheNode.setRead(jobClass, new DiscreteSampler(new Matrix(1, n).fill(1.0 / n)));
cacheNode.setItemSizes(sizes);
cacheNode.setCostCaps(caps);

    ... (4 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

routingMatrix.set(jobClass, jobClass, source, cacheNode, 1.0);
routingMatrix.set(hitClass, hitClass, cacheNode, sink, 1.0);
routingMatrix.set(missClass, missClass, cacheNode, sink, 1.0);
model.link(routingMatrix);

return model;
```

Running this edition prints

```
jline.lang.Network@28f67ac7
```

2.8.3 `cache_mmap_rr_env`

A marked MAP request stream into a random-replacement cache inside a random environment.

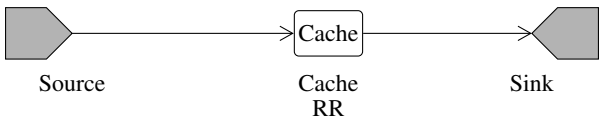

Figure 2.71: Topology of `cache_mmap_rr_env`: an open network with a cache node.

The example is built and solved as follows.

```
                                  jline.lang.processes.MarkedMAP mmap) {
Network model = new Network("MMAPCache");
```

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`.

```
int n = 4; // number of items
int m = 2; // cache capacity

Source source = new Source(model, "Source");
Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.RR);
Sink sink = new Sink(model, "Sink");

OpenClass rd1 = new OpenClass(model, "Read1", 0);
OpenClass rd2 = new OpenClass(model, "Read2", 0);
OpenClass hit1 = new OpenClass(model, "Hit1", 0);
OpenClass mis1 = new OpenClass(model, "Miss1", 0);
OpenClass hit2 = new OpenClass(model, "Hit2", 0);
OpenClass mis2 = new OpenClass(model, "Miss2", 0);

// Per-class item-popularity distributions (deliberately different)
Matrix p1 = new Matrix(1, n);   // class 1 favors low-index items
Matrix p2 = new Matrix(1, n);   // class 2 favors high-index items
double[] w1 = {8, 4, 2, 1};
for (int i = 0; i < n; i++) {
    p1.set(0, i, w1[i] / 15.0);
    p2.set(0, i, w1[n - 1 - i] / 15.0);
}
cacheNode.setRead(rd1, new DiscreteSampler(p1));

    ... (20 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

routingMatrix.set(mis1, mis1, cacheNode, sink, 1.0);
routingMatrix.set(hit2, hit2, cacheNode, sink, 1.0);
routingMatrix.set(mis2, mis2, cacheNode, sink, 1.0);
model.link(routingMatrix);

return model;
```

Running this edition prints

```
no no-arg static method cache_mmap_rr_env on jline.examples.java.basic.CacheModel
```

2.8.4 `cache_replc_climb`

The CLIMB policy, which promotes a hit by one list only.

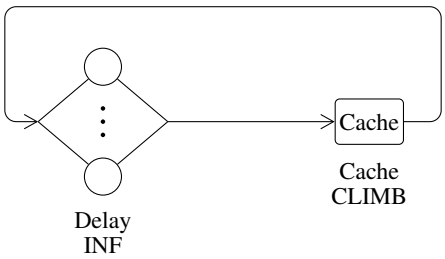

Figure 2.72: Topology of `cache_replc_climb`: a closed network over 1 queueing station with a cache node.

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`.

The example is built and solved as follows.

```
Network model = new Network("model");

int n = 5; // Number of items
int m = 2; // Cache capacity

Delay delay = new Delay(model, "Delay");
Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.CLIMB);

ClosedClass jobClass = new ClosedClass(model, "JobClass", 1, delay, 0);
ClosedClass hitClass = new ClosedClass(model, "HitClass", 0, delay, 0);
ClosedClass missClass = new ClosedClass(model, "MissClass", 0, delay, 0);

delay.setService(jobClass, new Exp(1.0));
cacheNode.setRead(jobClass, new Zipf(1.2, n));
cacheNode.setHitClass(jobClass, hitClass);
cacheNode.setMissClass(jobClass, missClass);

RoutingMatrix routingMatrix = model.initRoutingMatrix();
routingMatrix.set(jobClass, jobClass, delay, cacheNode, 1.0);
routingMatrix.set(hitClass, jobClass, cacheNode, delay, 1.0);
routingMatrix.set(missClass, jobClass, cacheNode, delay, 1.0);
model.link(routingMatrix);

return model;
```

Running this edition prints

```
jline.lang.Network@2ff4f00f
```

2.8.5 `cache_replc_fifo`

First-in first-out replacement.

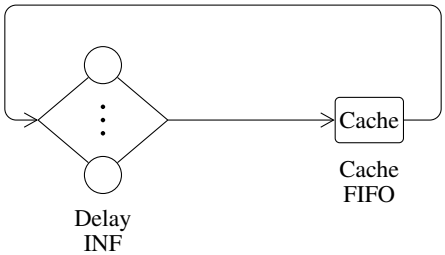

Figure 2.73: Topology of `cache_replc_fifo`: a closed network over 1 queueing station with a cache node.

The example is built and solved as follows.

```
public static void cache_replc_fifo() {
    //System.out.println("=== Cache FIFO Example ===");
    Network model = CacheModel.cache_replc_fifo();
```

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`.

```
    try {
        CTMC solver1 = new CTMC(model, "keep", false, "seed", 1);
        NetworkAvgNodeTable avgTable1 = solver1.getAvgNodeTable();
        //System.out.println("--- CTMC Solver ---");
        avgTable1.print();

        model.reset();
        SSA solver2 = new SSA(model, "samples", 100000, "verbose", true, "method", "serial",
    "seed", 23000);
        NetworkAvgNodeTable avgTable2 = solver2.getAvgNodeTable();
        //System.out.println("--- SSA Solver ---");
        avgTable2.print();

        model.reset();
        MVA solver3 = new MVA(model, "seed", 1);
        NetworkAvgNodeTable avgTable3 = solver3.getAvgNodeTable();
        //System.out.println("--- MVA Solver ---");
        avgTable3.print();

        model.reset();
        // The reference runs the RMF fluid limit here, which is what its golden
        // holds; the default fluid closure answers a different question.
        FLD solverFLD = new FLD(model, "method", "rmf");
        solverFLD.getAvgNodeTable().print();

    } catch (Exception e) {
        //System.out.println("Error in cache_replc_fifo: " + e.getMessage());
        e.printStackTrace();
    }
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node               JobClass   QLen  Util  RespT  ResidT  ArvR  Tput
Delay              JobClass      1     1      1       1     1     1
Cache              JobClass      0     0      0       0     1     0
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     0     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
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 100001.
Node               JobClass   QLen  Util  RespT  ResidT     ArvR     Tput
Delay              JobClass      1     1      1       1        1        1
Cache              JobClass      0     0      0       0        1        0
Cache              HitClass      0     0      0       0        0  0.39783
Cache              MissClass     0     0      0       0        0  0.60217
CS_Cache_to_Delay  JobClass      0     0      0       0        0        1

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

Node               JobClass   QLen  Util  RespT  ResidT        ArvR  Tput
Delay              JobClass      1     1      1       1           1     1
Cache              JobClass      0     0      0       0           1     0
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
```

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.

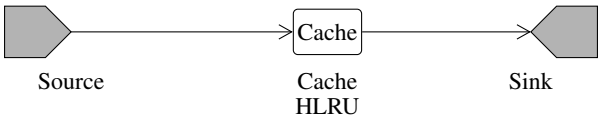

Figure 2.74: Topology of cache_replc_hlru: an open network with a cache node.

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.

```
Network model = new Network("model");

int n = 6; // Number of items
Matrix m = new Matrix(1, 2); // list 1 holds 2 items, list 2 holds 1
m.set(0, 0, 2);
m.set(0, 1, 1);

Source source = new Source(model, "Source");
Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.HLRU);
Sink sink = new Sink(model, "Sink");

OpenClass jobClass = new OpenClass(model, "InitClass", 0);
OpenClass hitClass = new OpenClass(model, "HitClass", 0);
OpenClass missClass = new OpenClass(model, "MissClass", 0);

source.setArrival(jobClass, new Exp(1.0));
cacheNode.setRead(jobClass, new Zipf(1.2, n));
cacheNode.setHitClass(jobClass, hitClass);
cacheNode.setMissClass(jobClass, missClass);

RoutingMatrix routingMatrix = model.initRoutingMatrix();
routingMatrix.set(jobClass, jobClass, source, cacheNode, 1.0);
routingMatrix.set(hitClass, hitClass, cacheNode, sink, 1.0);
routingMatrix.set(missClass, missClass, cacheNode, sink, 1.0);
model.link(routingMatrix);

return model;
```

Running this edition prints

```
jline.lang.Network@546a03af
```

2.8.7 cache_replc_lru

Least-recently-used replacement, with a closed class circulating between a Delay and the cache.
The example is built and solved as follows.

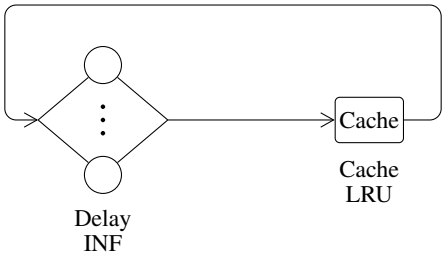

Figure 2.75: Topology of `cache_replc_lru`: a closed network over 1 queueing station with a cache node.

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
Solvers	CTMC, SSA, MVA

Table 2.75: Features of `cache_replc_lru`.

```
Network model = CacheModel.cache_replc_lru();

try {
    new CTMC(model, "keep", false).getAvgNodeTable().print();
    model.reset();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new SSA(model, "samples", 100000, "verbose", true, "method", "serial", "seed", 23000)
            .getAvgNodeTable().print();
    model.reset();
} catch (Exception e) {
    System.out.println("SSA failed: " + e.getMessage());
}
try {
    new MVA(model).getAvgNodeTable().print();
    model.reset();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node               JobClass   QLen  Util  RespT  ResidT  ArvR  Tput
Delay              JobClass      1     1      1       1     1     1
Cache              JobClass      0     0      0       0     1     0
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     0     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
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 100001.
Node               JobClass   QLen  Util  RespT  ResidT        ArvR     Tput
Delay              JobClass      1     1      1       1           1        1
Cache              JobClass      0     0      0       0           1        0
Cache              HitClass      0     0      0       0           0  0.40033
Cache              MissClass     0     0      0       0           0  0.59967
CS_Cache_to_Delay  JobClass      0     0      0       0  5.5511e-17        1

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

```
Node               JobClass    QLen  Util  RespT  ResidT  ArvR  Tput
Delay              JobClass       1     1      1       1     1     1
Cache              JobClass       0     0      0       0     1     0
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     0     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
```

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.

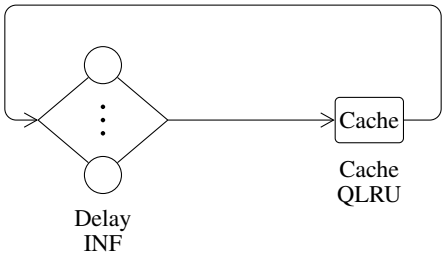

Figure 2.76: Topology of `cache_replc_qlru`: a closed network over 1 queueing station with a cache node.

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.

```
Network model = new Network("model");

int n = 5; // Number of items
int m = 2; // Cache capacity

Delay delay = new Delay(model, "Delay");
Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.QLRU);
cacheNode.setAdmissionProb(0.5); // admit a missed item with probability q=0.5

ClosedClass jobClass = new ClosedClass(model, "JobClass", 1, delay, 0);
ClosedClass hitClass = new ClosedClass(model, "HitClass", 0, delay, 0);
ClosedClass missClass = new ClosedClass(model, "MissClass", 0, delay, 0);

delay.setService(jobClass, new Exp(1.0));
cacheNode.setRead(jobClass, new Zipf(1.2, n));
cacheNode.setHitClass(jobClass, hitClass);
cacheNode.setMissClass(jobClass, missClass);

RoutingMatrix routingMatrix = model.initRoutingMatrix();
routingMatrix.set(jobClass, jobClass, delay, cacheNode, 1.0);
routingMatrix.set(hitClass, jobClass, cacheNode, delay, 1.0);
routingMatrix.set(missClass, jobClass, cacheNode, delay, 1.0);
model.link(routingMatrix);
```

```
return model;
```

Running this edition prints

```
jline.lang.Network@2ff4f00f
```

2.8.9 cache_replc_routing

Cache-aware routing: hits and misses leave the cache on different arcs, under RAND and round-robin routing.

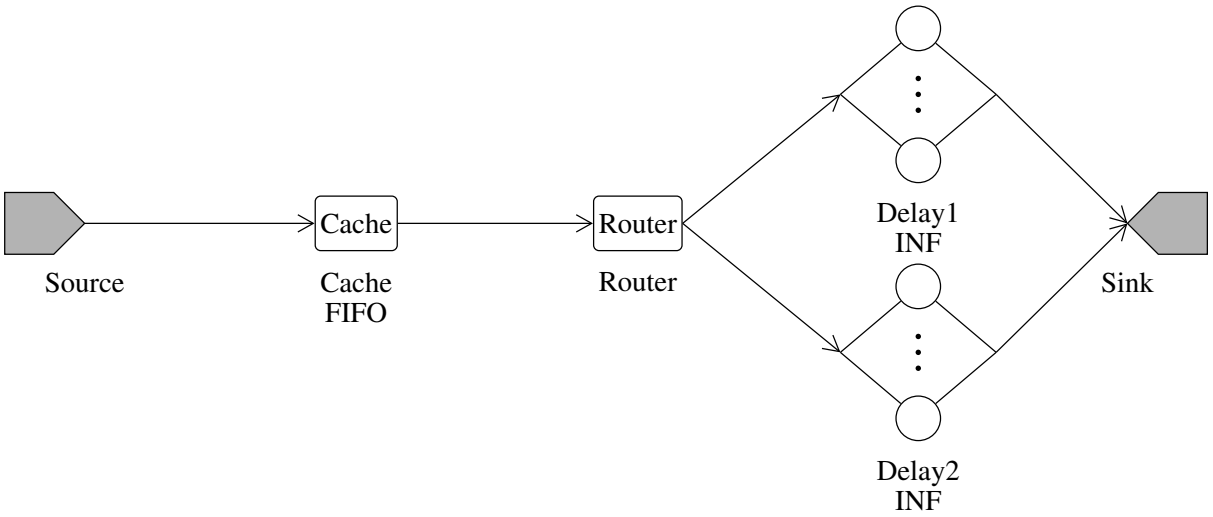

Figure 2.77: Topology of cache_replc_routing: an open network over 2 queueing stations with a cache node and a router.

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.

```
public static void cache_replc_routing() {
    //System.out.println("=== Cache Routing Example ===");
    Network model = CacheModel.cache_replc_routing();

    try {
        CTMC solver1 = new CTMC(model, "keep", false, "cutoff", 1, "seed", 1);
        NetworkAvgNodeTable avgTable1 = solver1.getAvgNodeTable();
        //System.out.println("--- CTMC Solver ---");
        avgTable1.print();

        model.reset();
        SSA solver2 = new SSA(model, "samples", 10000, "verbose", true, "method", "serial", "
    seed", 1);
        NetworkAvgNodeTable avgTable2 = solver2.getAvgNodeTable();
        //System.out.println("--- SSA Solver ---");
```

```
        avgTable2.print();

        model.reset();
        MVA solver3 = new MVA(model, "seed", 1);
        NetworkAvgNodeTable avgTable3 = solver3.getAvgNodeTable();
        //System.out.println("--- MVA Solver ---");
        avgTable3.print();

        model.reset();
        NC solver4 = new NC(model, "seed", 1);
        NetworkAvgNodeTable avgTable4 = solver4.getAvgNodeTable();
        //System.out.println("--- NC Solver ---");

    ... (12 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        //System.out.println("Miss Ratio: " + missRatio);

    } catch (Exception e) {
        //System.out.println("Error in cache_replc_routing: " + e.getMessage());
        e.printStackTrace();
    }
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
     truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
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.
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node    JobClass       QLen      Util  RespT  ResidT     ArvR     Tput
Source  InitClass         0         0      0       0        0   1.9808
Cache   InitClass         0         0      0       0   1.9808        0
Cache   HitClass          0         0      0       0        0  0.79232
Cache   MissClass         0         0      0       0        0   1.1885
Router  HitClass          0         0      0       0  0.79232  0.79232
Router  MissClass         0         0      0       0   1.1885   1.1885
Delay1  HitClass   0.038955  0.038955    0.1    0.02  0.39616  0.38955
Delay1  MissClass   0.42464   0.42464      1     0.3  0.59424  0.42464
Delay2  HitClass   0.020111  0.020111   0.05    0.01  0.39616  0.40223
Delay2  MissClass   0.29187   0.29187    0.5    0.15  0.59424  0.58373
Sink    HitClass          0         0      0       0  0.79232        0
Sink    MissClass         0         0      0       0   1.1885        0

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

Router  HitClass      0     0      0       0   0.8   0.8
Router  MissClass     0     0      0       0   1.2   1.2
Delay1  HitClass      0     0      0       0   0.4     0
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
```

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.

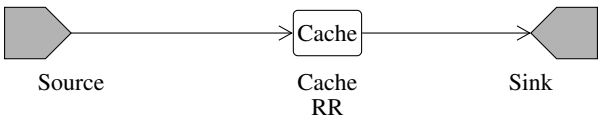

Figure 2.78: Topology of `cache_replc_rr`: an open network with a cache node.

The example is built and solved as follows.

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`.

```
public static void cache_replc_rr() {
    //System.out.println("=== Cache RR Example ===");
    Network model = CacheModel.cache_replc_rr();

    try {
        CTMC solver1 = new CTMC(model, "keep", false, "cutoff", 1, "seed", 1);
        NetworkAvgNodeTable avgTable1 = solver1.getAvgNodeTable();
        //System.out.println("--- CTMC Solver ---");
        avgTable1.print();

        model.reset();
        SSA solver2 = new SSA(model, "samples", 10000, "verbose", true, "method", "serial", "
    seed", 1);
        NetworkAvgNodeTable avgTable2 = solver2.getAvgNodeTable();
        //System.out.println("--- SSA Solver ---");
        avgTable2.print();

        model.reset();
        MVA solver3 = new MVA(model, "seed", 1);
        NetworkAvgNodeTable avgTable3 = solver3.getAvgNodeTable();
        //System.out.println("--- MVA Solver ---");
        avgTable3.print();

        model.reset();
        NC solver4 = new NC(model, "seed", 1);
        NetworkAvgNodeTable avgTable4 = solver4.getAvgNodeTable();
        //System.out.println("--- NC Solver ---");

    ... (12 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        //System.out.println("Miss Ratio: " + missRatio);

    } catch (Exception e) {
        //System.out.println("Error in cache_replc_rr: " + e.getMessage());
        e.printStackTrace();
    }
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
     truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node    JobClass   QLen  Util  RespT  ResidT     ArvR     Tput
Source  InitClass     0     0      0       0        0        2
Cache   InitClass     0     0      0       0        2        0
Cache   HitClass      0     0      0       0        0   1.1808
Cache   MissClass     0     0      0       0        0  0.81921
Sink    HitClass      0     0      0       0   1.1808        0
Sink    MissClass     0     0      0       0  0.81921        0
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 10001.
Node    JobClass   QLen  Util  RespT  ResidT     ArvR     Tput
Source  InitClass     0     0      0       0        0        2
Cache   InitClass     0     0      0       0        2        0
Cache   HitClass      0     0      0       0        0   1.1633
Cache   MissClass     0     0      0       0        0  0.83669
Sink    HitClass      0     0      0       0   1.1633        0

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

Fluid analysis [method: rmf; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
Node    JobClass   QLen  Util  RespT  ResidT     ArvR     Tput
```

```
Source  InitClass     0     0      0       0        0        2
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
```

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.

```
int n = 10;              // number of items
int[] m = {3, 2};        // list capacities (2-list cache)
double alpha = 0.8;      // Zipf exponent

double[] p = new double[n];
double sum = 0.0;
for (int i = 0; i < n; i++) {
    p[i] = Math.pow(i + 1, -alpha);
    sum += p[i];
}
for (int i = 0; i < n; i++) p[i] /= sum;

System.out.println("Cache parameters: n=" + n + ", m=[" + m[0] + ", " + m[1]
        + "], Zipf(" + alpha + ")");

CacheRMF rmf = new CacheRMF(p, m);

Object[] ss = rmf.meanFieldExpansionSteadyState(1);
double[] pi = (double[]) ss[0];
double[] V = (double[]) ss[1];
double[] piRefined = new double[pi.length];
for (int i = 0; i < pi.length; i++) piRefined[i] = pi[i] + V[i] / n;

System.out.println("\nSteady-state results (refined mean field):");
double totalHit = 0.0;
for (int k = 1; k <= m.length; k++) {

    ... (25 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    double hr = 0.0;
    for (int k = 1; k <= m.length; k++) hr += rmf.hitRate(xt, k);
    jline.io.LineResultRecorder.scalar("CACHE",
            String.format("t=%.3f", T[idx]), "Transient", hr);
    System.out.printf("  t=%7.3f: hit_rate=%.6f%n", T[idx], hr);
}
```

Running this edition prints

```
Cache parameters: n=10, m=[3, 2], Zipf(0.8)

Steady-state results (refined mean field):
  Hit rate (list 1): 0.312854
  Hit rate (list 2): 0.301606
  Miss rate:         0.385540
  Total hit prob:    0.614460
  Total miss prob:   0.385540

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.

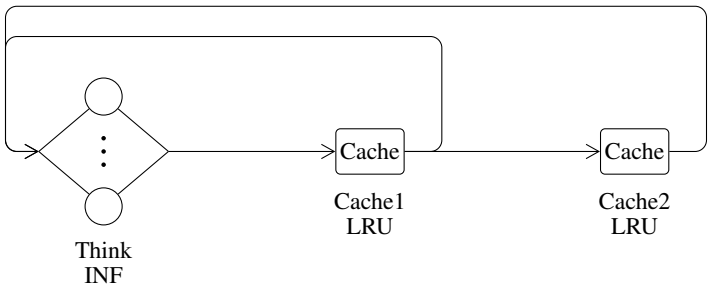

Figure 2.79: Topology of `cachenet_tandem`: a closed network over 1 queueing station with a cache node.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.8.13 `retrieval_chain`

A chain of caches, each backing the one before it.
The example is built and solved as follows.

```
Matrix serviceRates = new Matrix(new double[][] {
        {2.0, 2.0, 2.0},  // Queue_1 per-item rates
        {3.0, 3.0, 3.0}   // Queue_2 per-item rates
```

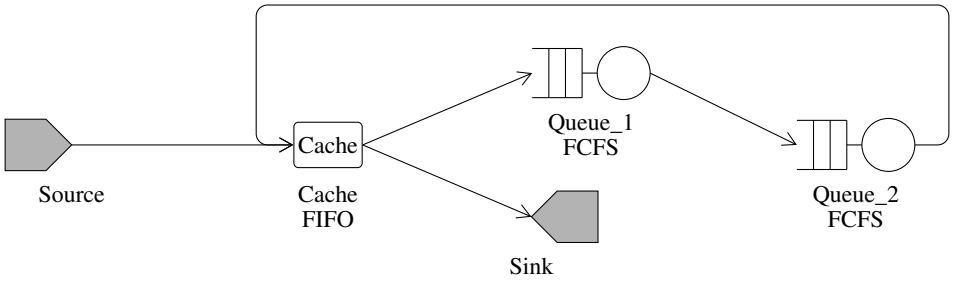

Figure 2.80: Topology of `retrieval_chain`: an open network over 2 queueing stations with a cache node.

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`.

```
});
Network model = CacheRetrievalSystemModel.chain_retrieval_system_model(
        new double[] {0.6, 0.3, 0.1}, serviceRates, 2, "[1]", SchedStrategy.FCFS);
solveAll(model);
```

Running this edition prints

```
WARNING: [solver_ssa_analyzer_serial] Retrieval-system expected latency is not currently implemented;
     reporting NaN.
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 100001.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.43354         0.14244   0.42402  0.43354         0.14244
     0.42402  0.56646     NaN       NaN
LDES events:  20000   40000  60000  80000 100000 120000 140000 160000 180000 200000 220000 240000 260000
     280000 300000 320000 340000 360000 380000 400000 420000 440000 460000 480000 500000 520000 540000
     560000 580000 600000 620000 640000 660000 680000 700000 720000 740000 760000 780000 800000 820000
     840000 860000 880000 900000 920000 940000 960000 980000 1000000 1000000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 1000000.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.43802         0.14432   0.41766  0.43802         0.14432
     0.41766  0.56198  0.83647       NaN
MVA analysis [method: default/fpi; type: approximate, deterministic; lang: java; 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  0.40688         0.16346   0.42966  0.40688         0.16346
     0.42966  0.59312  0.87138       NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: java; 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
```

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.

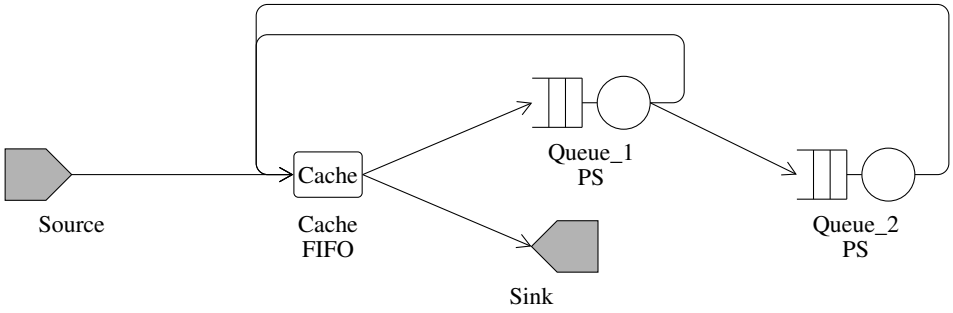

Figure 2.81: Topology of `retrieval_default`: an open network over 2 queueing stations with a cache node.

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`.

The example is built and solved as follows.

```
Network model = CacheRetrievalSystemModel.retrieval_default();
solveAll(model);
```

Running this edition prints

```
WARNING: [solver_ssa_analyzer_serial] Retrieval-system expected latency is not currently implemented;
     reporting NaN.
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 100001.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        1      3  0.46222        0.076989   0.46079  0.46222        0.076989
     0.46079  0.53778     NaN       NaN
LDES events:  20000   40000  60000  80000 100000 120000 140000 160000 180000 200000 220000 240000 260000
     280000 300000 320000 340000 360000 380000 400000 420000 440000 460000 480000 500000 520000 540000
     560000 580000 600000 620000 640000 660000 680000 700000 720000 740000 760000 780000 800000 820000
     840000 860000 880000 900000 920000 940000 960000 980000 1000000 1000000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
      Iterations: 1000000.
```

```
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        1      3  0.46348        0.078748   0.45777  0.46348        0.078748
     0.45777  0.53652  0.57635       NaN
MVA analysis [method: default/fpi; type: approximate, deterministic; lang: java; 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  0.42401        0.084524   0.49147  0.42401        0.084524
     0.49147  0.57599  0.54262       NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: java; 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
```

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.

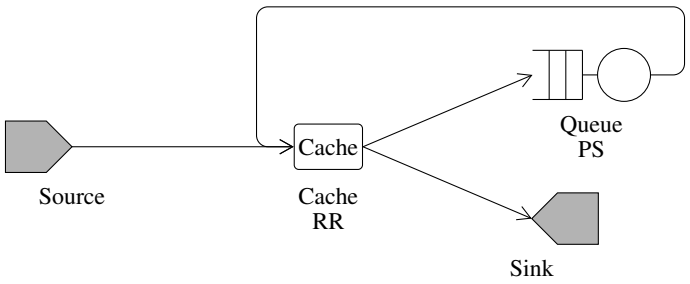

Figure 2.82: Topology of `retrieval_ps`: an open network over 1 queueing station with a cache node.

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.

```
double[] accessProb = {49, 49, 49, 49, 7, 1, 1};
double total = 0.0;
for (double p : accessProb) total += p;
```

```
for (int i = 0; i < accessProb.length; i++) accessProb[i] /= total;    // lambda =
    (49,49,49,49,7,1,1)/205
Network model = CacheRetrievalSystemModel.simple_retrieval_system_model(
        accessProb,
        new double[] {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},              // mu_i = 1
        "[6]", SchedStrategy.PS, ReplacementStrategy.RR);
solveAll(model);
```

Running this edition prints

```
WARNING: [solver_ssa_analyzer_serial] Retrieval-system expected latency is not currently implemented;
    reporting NaN.
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
    Iterations: 100001.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR  ResidT  ListCost
Cache  InitClass     0        6      7  0.98494       0.0014294  0.013632  0.98494       0.0014294
    0.013632  0.015061     NaN       NaN
LDES events:  20000   40000  60000  80000 100000 120000 140000 160000 180000 200000 220000 240000 260000
    280000 300000 320000 340000 360000 380000 400000 420000 440000 460000 480000 500000 520000 540000
    560000 580000 600000 620000 640000 660000 680000 700000 720000 740000 760000 780000 800000 820000
    840000 860000 880000 900000 920000 940000 960000 980000 1000000 1000000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 1000000.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate      ArvR   ResidT  ListCost
Cache  InitClass     0        6      7  0.98298        0.002025  0.014991  0.98298        0.002025
    0.014991  0.017016  0.99256       NaN
MVA analysis [method: default/fpi; type: approximate, deterministic; lang: java; 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  0.97573       0.0034469  0.020826  0.97573       0.0034469
    0.020826  0.024273  1.0155       NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: java; 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
```

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.

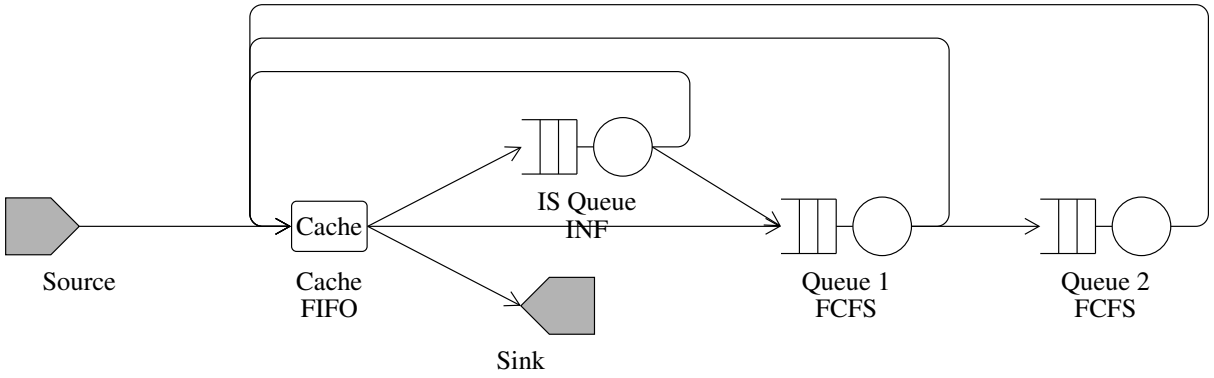

Figure 2.83: Topology of `retrieval_routing`: an open network over 3 queueing stations with a cache node.

The example is built and solved as follows.

```
Matrix serviceRates = new Matrix(new double[][] {
        {2.0, 2.0, 2.0},  // IS queue
```

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`.

```
        {3.0, 3.0, 3.0},  // Queue 1
        {3.0, 3.0, 3.0}   // Queue 2
});
Network model = CacheRetrievalSystemModel.retrieval_system_with_probabilistic_routing(
        new double[] {0.6, 0.3, 0.1}, serviceRates, "[2]", SchedStrategy.FCFS);
solveAll(model);
```

Running this edition prints

```
WARNING: [solver_ssa_analyzer_serial] Retrieval-system expected latency is not currently implemented;
    reporting NaN.
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
    Iterations: 100001.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR  ResidT  ListCost
Cache  InitClass     0        2      3  0.78896        0.038947   0.17209  0.78896        0.038947
    0.17209  0.21104     NaN       NaN
LDES events:  20000   40000  60000  80000 100000 120000 140000 160000 180000 200000 220000 240000 260000
    280000 300000 320000 340000 360000 380000 400000 420000 440000 460000 480000 500000 520000 540000
    560000 580000 600000 620000 640000 660000 680000 700000 720000 740000 760000 780000 800000 820000
    840000 860000 880000 900000 920000 940000 960000 980000 1000000 1000000
LDES analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 1000000.
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
    MissRate     ArvR   ResidT  ListCost
Cache  InitClass     0        2      3  0.78284        0.039019   0.17814  0.78284        0.039019
    0.17814  0.21716  0.61714       NaN
MVA analysis [method: default/fpi; type: approximate, deterministic; lang: java; 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  0.74517        0.050821   0.20401  0.74517        0.050821
    0.20401  0.25483  0.62901       NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: java; 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
```

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.

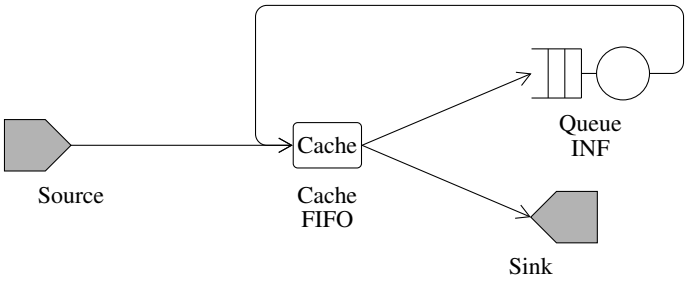

Figure 2.84: Topology of `retrieval_simple`: an open network over 1 queueing station with a cache node.

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.

```
Network model = CacheRetrievalSystemModel.simple_retrieval_system_model(
        new double[] {0.6, 0.3, 0.1},
        new double[] {2.0, 2.0, 2.0},
        "[1]", SchedStrategy.INF);
solveCtmc(model);
solveAll(model);
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 1 for open/mixed model. State space
     truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
WARNING: [solver_ctmc_analyzer] Retrieval-system expected latency is not currently implemented; reporting
     NaN.
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node    JobClass   QLen  Util  RespT  ResidT     ArvR     Tput
Source  InitClass     0     0      0       0        0        1
Cache   InitClass     0     0      0       0        1        0
Cache   HitClass      0     0      0       0        0  0.51077
Cache   MissClass     0     0      0       0        0  0.48923
Sink    HitClass      0     0      0       0  0.51077        0
Sink    MissClass     0     0      0       0  0.48923        0
CTMC  hit=[ 0.510772584520104   NaN   NaN   NaN   NaN   NaN ]
 miss=[ 0.489227415479896   NaN   NaN   NaN   NaN   NaN ]
 expectedLatency=[ NaN   NaN   NaN   NaN   NaN   NaN ]

WARNING: [solver_ssa_analyzer_serial] Retrieval-system expected latency is not currently implemented;
     reporting NaN.
SSA analysis [method: serial; type: approximate, randomized; lang: java; env: python] completed in <t>s.
     Iterations: 100001.

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

```
Node   JobClass   List  ListCap  Items  HitProb  DelayedHitProb  MissProb  HitRate  DelayedHitRate
     MissRate    ArvR  ResidT  ListCost
Cache  InitClass     0        1      3   0.4446        0.091824   0.46358   0.4446        0.091824
     0.46358  0.5554  0.4997       NaN
MVA analysis [method: default/fpi; type: approximate, deterministic; lang: java; 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  0.41334         0.10048   0.48618  0.41334         0.10048
     0.48618  0.58666     0.5       NaN
NC analysis [method: default/exact; type: exact, deterministic; lang: java; 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
```

The rows repeat for each of the 5 solvers the script runs (CTMC, 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.

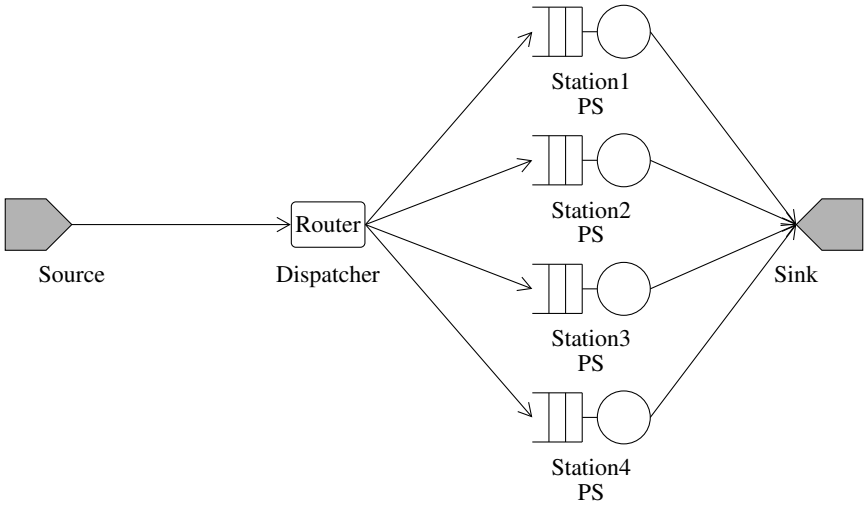

Figure 2.85: Topology of `cl_basic`: an open network over 4 queueing stations with a router.

The example is built and solved as follows.

```
Matrix lambda = new Matrix(1, 1);
lambda.set(0, 0, 0.4);
Matrix D = new Matrix(4, 1);
for (int i = 0; i < 4; i++) D.set(i, 0, 1.0);  // mean service time = 1
return Network.clusterPs(lambda, D, RoutingStrategy.RAND);
```

Running this edition prints

```
jline.lang.Network@69d0a921
```

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`.

2.9.2 `cl_closed`

The closed version, with a fixed population of requests.

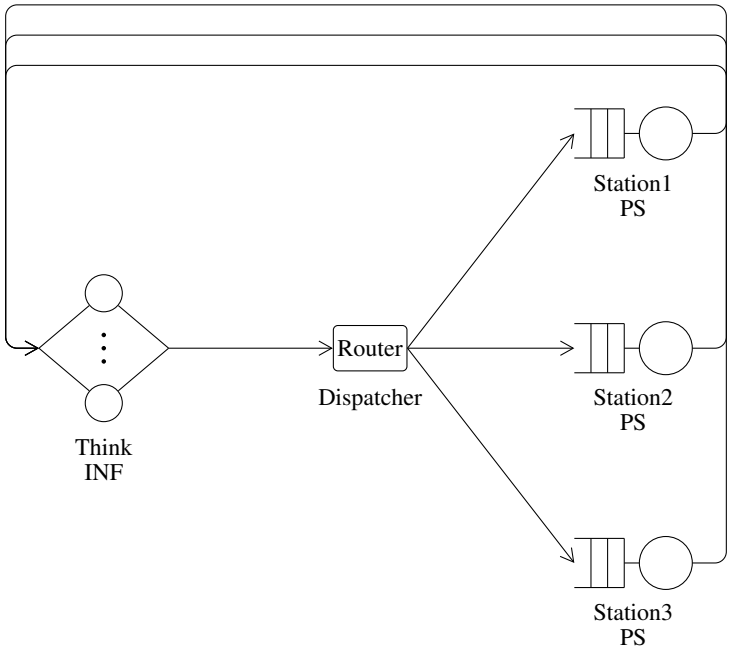

Figure 2.86: Topology of `cl_closed`: a closed network over 4 queueing stations with a router.

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.

```
Matrix N = new Matrix(1, 1);
N.set(0, 0, 8);
Matrix Z = new Matrix(1, 1);
Z.set(0, 0, 1.0);  // think time
Matrix D = new Matrix(3, 1);
for (int i = 0; i < 3; i++) D.set(i, 0, 1.0);
Matrix S = new Matrix(3, 1, 3);
```

```
S.fill(1.0);
SchedStrategy[] sched = new SchedStrategy[3];
for (int i = 0; i < 3; i++) sched[i] = SchedStrategy.PS;
return Network.clusterClosed(N, Z, D, sched, S, RoutingStrategy.RAND);
```

Running this edition prints

```
jline.lang.Network@3eb07fd3
```

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.

```
System.out.println("\n--- ex2_compare_dispatching: RAND vs RROBIN vs JSQ ---");
Cluster cluster = new Cluster().setNumStations(4).setArrivalRate(1.0).setServiceRate(0.4)
        .setScheduling(SchedStrategy.PS);
// RROBIN and JSQ routing are outside the MVA feature set, so the
// dispatching comparison uses the LDES simulator, which supports all three
Map<RoutingStrategy, NetworkAvgTable> results = cluster.compareDispatching(
        SolverLDES.class,
        RoutingStrategy.RAND,
        RoutingStrategy.RROBIN,
        RoutingStrategy.JSQ);
for (Map.Entry<RoutingStrategy, NetworkAvgTable> e : results.entrySet()) {
    System.out.println("\nDispatching = " + e.getKey());
    e.getValue().print();
}
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.9.4 cl_multiclass

Several request classes through one cluster.

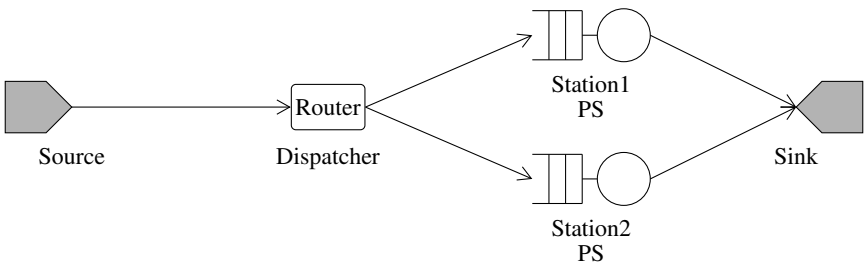

Figure 2.87: Topology of `cl_multiclass`: an open network over 2 queueing stations with a router.

The example is built and solved as follows.

```
Matrix lambda = new Matrix(1, 2);
lambda.set(0, 0, 0.3);
```

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`.

```
lambda.set(0, 1, 0.2);
Matrix D = new Matrix(2, 2);
D.set(0, 0, 1.0); D.set(0, 1, 0.5);
D.set(1, 0, 1.0); D.set(1, 1, 0.5);
Matrix S = new Matrix(2, 1, 2);
S.fill(1.0);
return Network.clusterPs(lambda, D, S, RoutingStrategy.RAND);
```

Running this edition prints

```
jline.lang.Network@446cdf90
```

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.

```
System.out.println("\n--- cl_scheduling: FCFS against PS at SCV 4 ---");
Cluster cluster = new Cluster().setNumStations(3).setArrivalRate(0.9).setServiceRate(0.5)
        .setDispatching(RoutingStrategy.RAND);
cluster.setServiceSCV(4.0);
Map<SchedStrategy, NetworkAvgTable> results = cluster.compareScheduling(
        model -> new SolverSSA(model, "seed", 23000, "samples", 20000).getAvgTable(),
        SchedStrategy.FCFS, SchedStrategy.PS);
for (Map.Entry<SchedStrategy, NetworkAvgTable> e : results.entrySet()) {
    System.out.println("\nScheduling = " + e.getKey());
    e.getValue().print();
}
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.
The example is built and solved as follows.

Solvers	MVA

Table 2.90: Features of `cl_stations`.

```
System.out.println("\n--- cl_stations: sweep the number of servers ---");
Cluster cluster = new Cluster().setNumStations(1).setArrivalRate(1.6).setServiceRate(1.0)
        .setScheduling(SchedStrategy.PS)
        .setDispatching(RoutingStrategy.RAND);
Map<Integer, NetworkAvgTable> sweep = cluster.sweepNumStations(
        new int[]{2, 3, 4, 6}, SolverMVA.class);
for (Map.Entry<Integer, NetworkAvgTable> e : sweep.entrySet()) {
    System.out.println("\nstations = " + e.getKey());
    e.getValue().print();
}
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

```
System.out.println("\n--- ex6_sweep: arrival-rate sweep ---");
Cluster cluster = new Cluster().setNumStations(2).setArrivalRate(0.1).setServiceRate(1.0)
        .setScheduling(SchedStrategy.PS)
        .setDispatching(RoutingStrategy.RAND);
Map<Double, NetworkAvgTable> sweep = cluster.sweepArrivalRate(
        new double[]{0.2, 0.5, 0.9, 1.5}, SolverMVA.class);
for (Map.Entry<Double, NetworkAvgTable> e : sweep.entrySet()) {
    System.out.println("\nlambda = " + e.getKey());
    e.getValue().print();
}
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.9.8 `dispatch_closed`

The closed version.

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`.

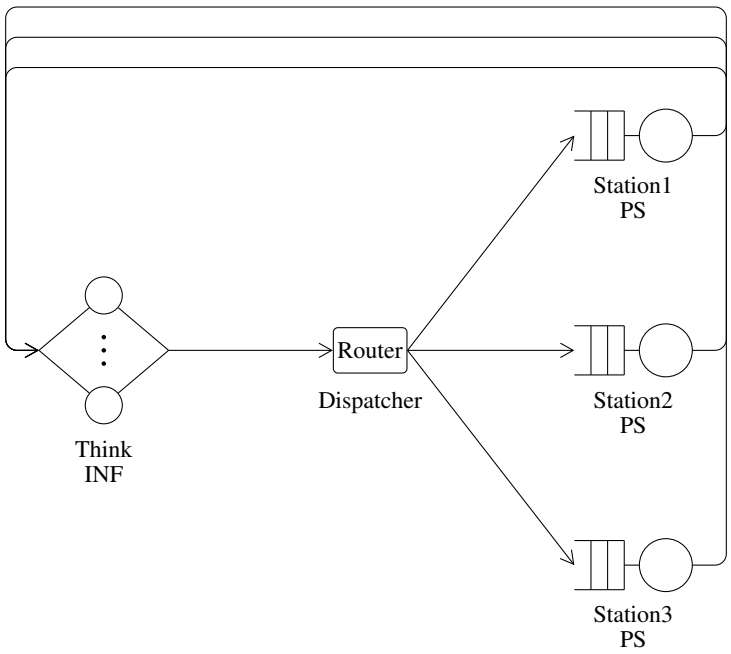

Figure 2.88: Topology of `dispatch_closed`: a closed network over 4 queueing stations with a router.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.9.10 `dispatch_open`

The dispatcher idiom written out in full, open workload.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

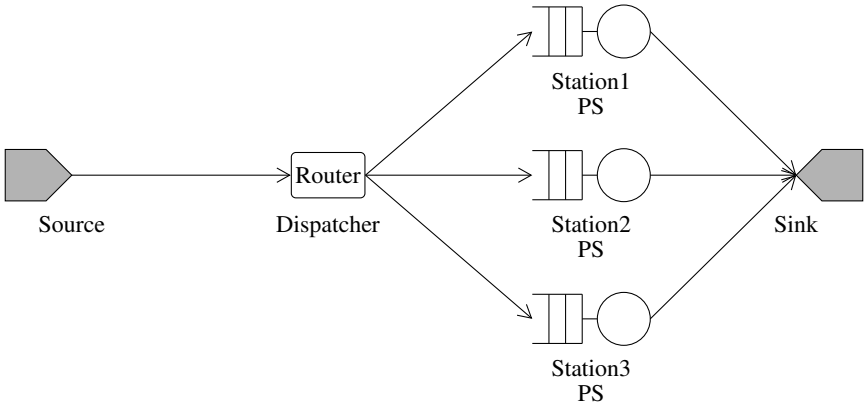

Figure 2.89: Topology of `dispatch_open`: an open network over 3 queueing stations with a router.

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`.

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.
The example is built and solved as follows.

```
    public static void spn_basic_closed() throws Exception {
        Network model = StochPetriNetModel.spn_basic_closed();

        // CTMC solver (exact for small Petri nets)
//        try {
```

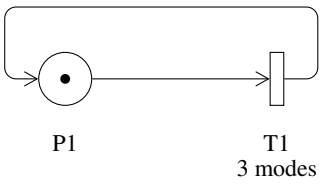

Figure 2.90: Topology of `spn_basic_closed`: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; a transition offering several firing modes is named with their 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`.

```
//             CTMC solverCtmc = new CTMC(model);
//             solverCtmc.getAvgTable().print();
//         } catch (Exception e) {
//             System.out.println("CTMC solver error: " + e.getMessage());
//         }

        // JMT solver (simulation)
        try {
            JMT solverJmt = new JMT(model, "seed", 23000);
            solverJmt.getAvgTable().print();
        } catch (Exception e) {
            System.out.println("JMT solver not available: " + e.getMessage());
        }
        pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

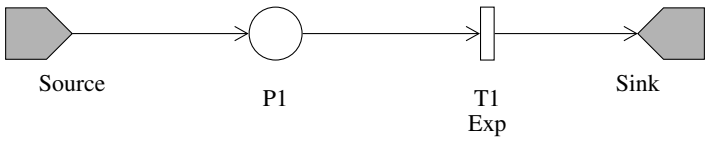

Figure 2.91: Topology of `spn_basic_open`: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

The example is built and solved as follows.

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`.

```
Network model = StochPetriNetModel.spn_basic_open();
JMT solver = new JMT(model, "seed", 23000);

solver.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

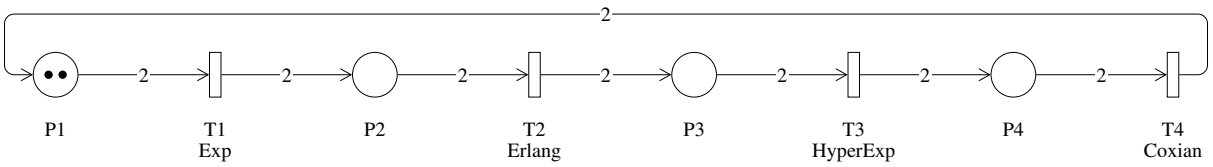

Figure 2.92: Topology of `spn_closed_fourplaces`: a stochastic Petri net over 4 places and 4 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

Nodes	8: Place $\times$ 4, Transition $\times$ 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`.

The example is built and solved as follows.

```
Network model = StochPetriNetModel.spn_closed_fourplaces();

// CTMC solver for exact solution
try {
    CTMC solverCtmc = new CTMC(model);
    solverCtmc.getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC solver error: " + e.getMessage());
}
```

```
// JMT solver for simulation
try {
    JMT solverJmt = new JMT(model, "seed", 23000);
    solverJmt.getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
P1       Class1    0.28846  0.28846      0.5      0.5  0.57692  0.57692
P2       Class1    0.76923  0.76923   1.3333   1.3333  0.57692  0.57692
P3       Class1       0.25     0.25  0.43333  0.43333  0.57692  0.57692
P4       Class1    0.69231  0.69231      1.2      1.2  0.57692  0.57692
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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

2.10.4 `spn_closed_twoplaces`

A closed net over two places with Erlang firing times.

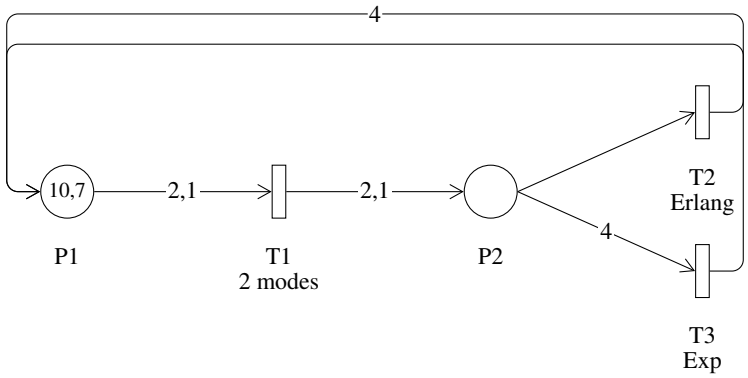

Figure 2.93: Topology of `spn_closed_twoplaces`: a stochastic Petri net over 2 places and 3 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity; 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`.

The example is built and solved as follows.

```
Network model = StochPetriNetModel.spn_closed_twoplaces();

// CTMC solver for exact solution
try {
    CTMC solverCtmc = new CTMC(model);
    solverCtmc.getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC solver error: " + e.getMessage());
}

// JMT solver for simulation
try {
    JMT solverJmt = new JMT(model, "seed", 23000);
    solverJmt.getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT solver not available: " + e.getMessage());
}
pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen    Util   RespT  ResidT    ArvR    Tput
P1       Class1    0.8893  0.8893  1.1857  1.1857    0.75    0.75
P1       Class2    1.6859  1.6859  1.0928  1.0928  1.5427  1.5427
P2       Class1    9.1107  9.1107  12.148  12.148    0.75    0.75
P2       Class2    5.3141  5.3141  3.4446  3.4446  1.5427  1.5427
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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

2.10.5 `spn_fluid_dae`

Fluid DAE analysis of stochastic Petri nets, including invariant constraints, immediate-transition flows, bounded markings and marking covariances.

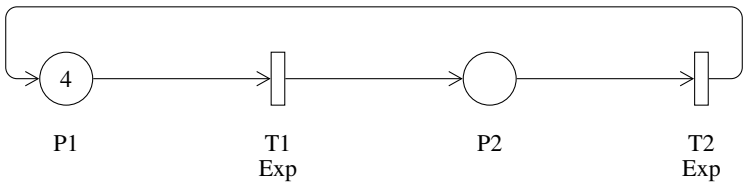

Figure 2.94: Topology of `spn_fluid_dae`: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.10.6 spn_fourmodes

Four firing modes, and the mode selection probabilities.

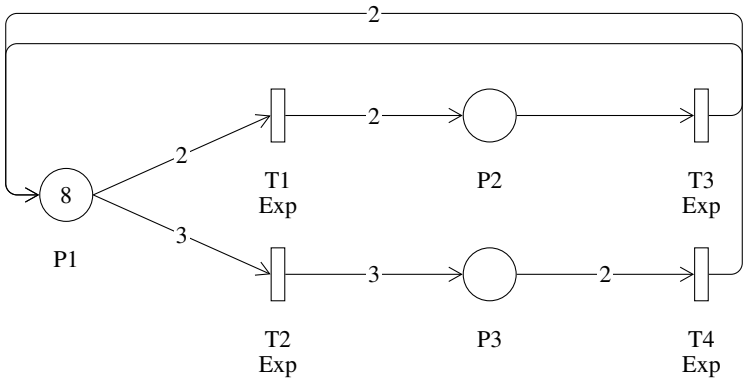

Figure 2.95: Topology of `spn_fourmodes`: a stochastic Petri net over 3 places and 4 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

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.

```
Network model = StochPetriNetModel.spn_fourmodes();
JMT solver = new JMT(model, "seed", 23000);

solver.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

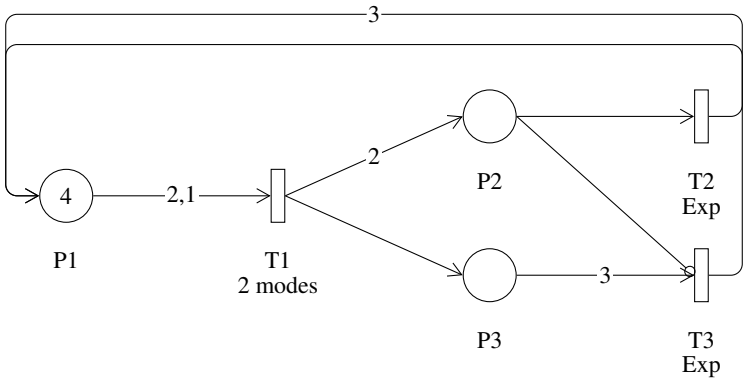

Figure 2.96: Topology of `spn_inhibiting`: a stochastic Petri net over 3 places and 3 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity; 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.

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`.

```
Network model = StochPetriNetModel.spn_inhibiting();
JMT solver = new JMT(model, "seed", 23000);

solver.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run

Solvers	CTMC

Table 2.102: Features of `spn_lpbounds`.

the identified implementation to obtain its results.

2.10.9 `spn_open_sevenplaces`

A seven-place open net, large enough that the state space matters.

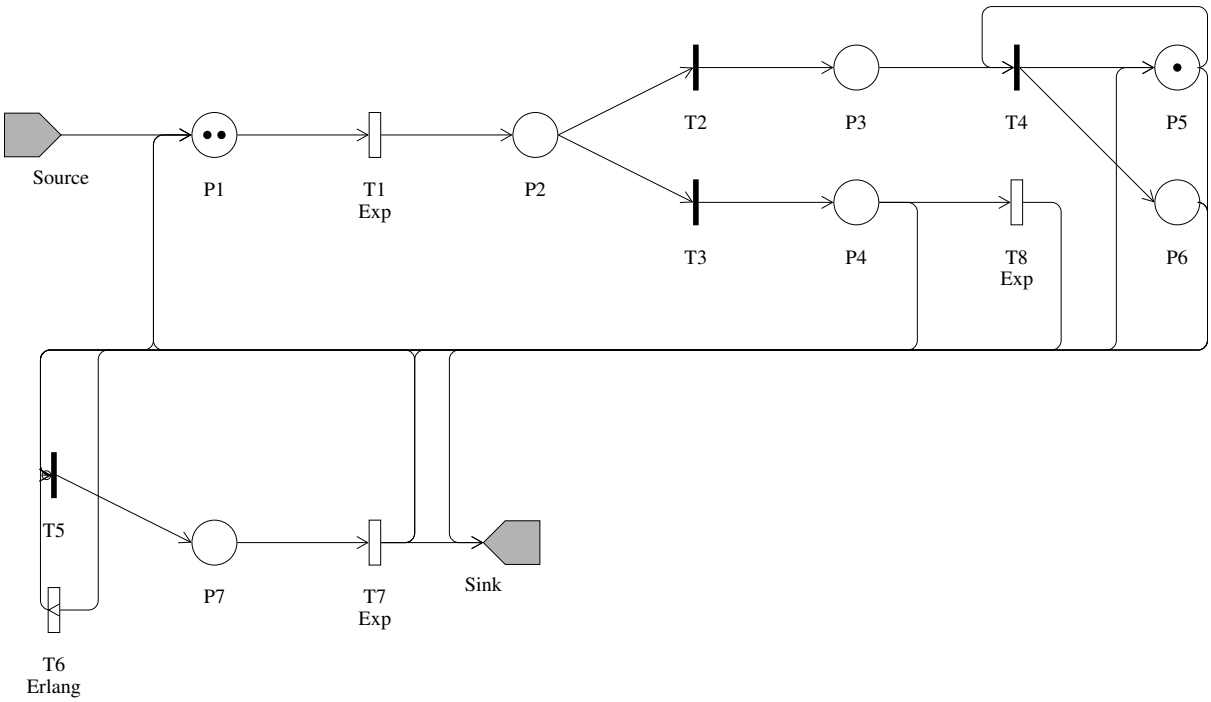

Figure 2.97: Topology of `spn_open_sevenplaces`: a stochastic Petri net over 7 places and 8 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the arc ending in a circle is an inhibitor and holds 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.

```
Network model = StochPetriNetModel.spn_open_sevenplaces();
JMT solver = new JMT(model, "seed", 23000);

solver.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

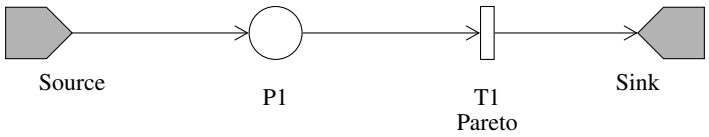

Figure 2.98: Topology of spn_pareto_service: a stochastic Petri net over 1 place and 1 transition; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

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.

The example is built and solved as follows.

```
Network model = StochPetriNetModel.spn_pareto_service();

try {
    new jline.solvers.wrappers.jmt.JMT(model, "seed", 23000, "samples", 10000)
            .getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.10.12 spn_queueing_place

A queueing station and a Petri net in one model, the two formalisms side by side.

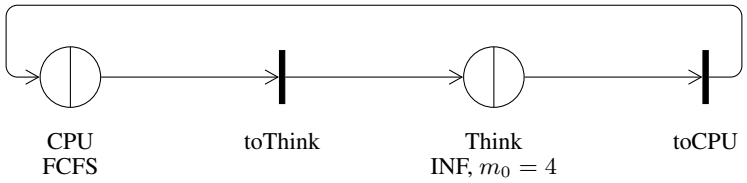

Figure 2.99: Topology of `spn_queueing_place`: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the 2 bisected circles are queueing places, whose tokens 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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

2.10.13 spn_twomodes

A transition with two firing modes.
The example is built and solved as follows.

```
Network model = StochPetriNetModel.spn_twomodes();
JMT solver = new JMT(model, "seed", 23000);
```

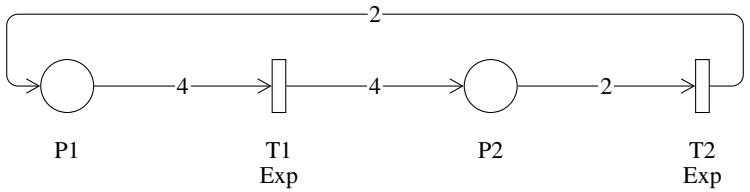

Figure 2.100: Topology of `spn_twomodes`: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; an arc that moves more than one token carries its multiplicity.

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`.

```
solver.getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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 `jline.examples.java.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.

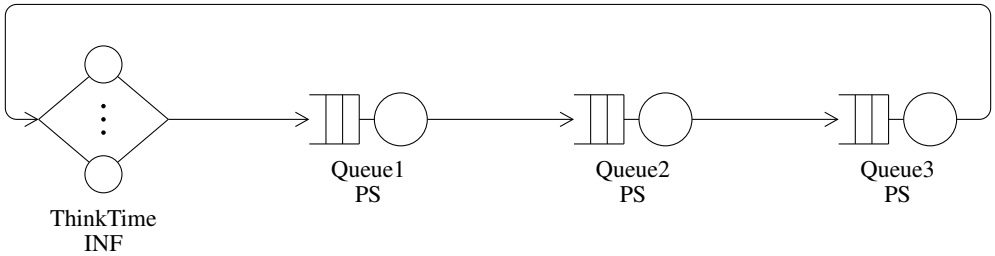

Figure 3.1: Topology of `fes_aggregation`: a closed network over 4 queueing stations.

The example is built and solved as follows.

```
Network model = LoadDependentModel.fes_aggregation();
System.out.println("MVA (original):");
new MVA(model, "method", "exact").getAvgTable().print();

FESResult fes = ModelAdapter.aggregateFES(
```

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`.

```
        model, subsetOf(model, new String[] {"Queue1", "Queue2"}));
System.out.println("MVA (FES model):");
new MVA(fes.fesModel, "method", "exact").getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
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
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA (FES model):
SEVERE: [getAvg] RuntimeException upon running runAnalyzer(): Exact class-dependent solver not available
     in MVA.
SEVERE: [getAvgTableImpl] Unable to compute results and therefore unable to print AvgTable: [getAvg]
     RuntimeException upon running runAnalyzer(): Exact class-dependent solver not available in MVA.
FAILED fes_aggregation: jline.io.LineException: [getAvgTableImpl] Unable to compute results and therefore
     unable to print AvgTable: [getAvg] RuntimeException upon running runAnalyzer(): Exact class-dependent
      solver not available in MVA.
```

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.
The example is built and solved as follows.

```
Network model = LoadDependentModel.fes_single_class();
```

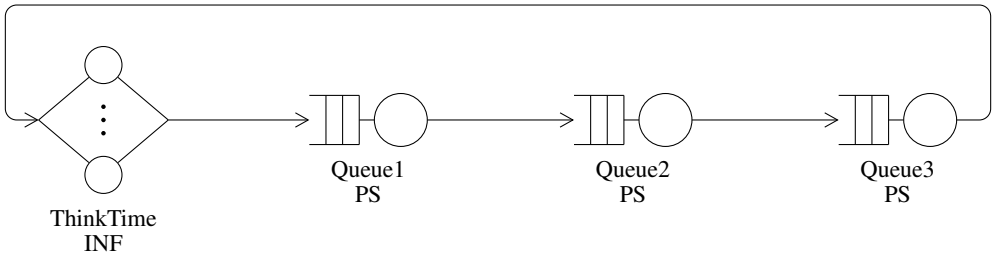

Figure 3.2: Topology of `fes_single_class`: a closed network over 4 queueing stations.

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`.

```
System.out.println("MVA (original):");
new MVA(model, "method", "exact").getAvgTable().print();

FESResult fes = ModelAdapter.aggregateFES(
        model, subsetOf(model, new String[] {"Queue1", "Queue2"}));
System.out.println("MVA (FES model):");
new MVA(fes.fesModel, "method", "exact").getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
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
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA (FES model):
SEVERE: [getAvg] RuntimeException upon running runAnalyzer(): Exact class-dependent solver not available
    in MVA.
SEVERE: [getAvgTableImpl] Unable to compute results and therefore unable to print AvgTable: [getAvg]
    RuntimeException upon running runAnalyzer(): Exact class-dependent solver not available in MVA.
FAILED fes_single_class: jline.io.LineException: [getAvgTableImpl] Unable to compute results and therefore
     unable to print AvgTable: [getAvg] RuntimeException upon running runAnalyzer(): Exact class-
    dependent solver not available in MVA.
```

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.
The example is built and solved as follows.

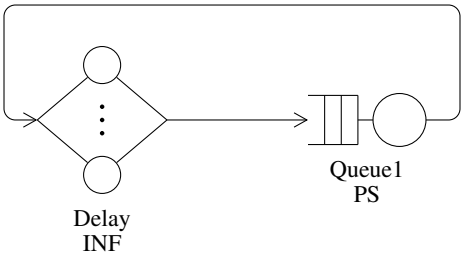

Figure 3.3: Topology of `ld_class_dependence`: a closed network over 2 queueing stations.

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`.

```
Network model = LoadDependentModel.ld_class_dependence();

// JMT is not solved here, as in ld_class_dependence.m: the JSIM writer
// has no representation for the class-dependence handle, so SolverJMT
// rejects the model rather than silently solving it unscaled.
new CTMC(model, "exact").getAvgTable().print();
new MVA(model, "method", "qd").getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: exact; type: exact, deterministic; lang: java; 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 analysis [method: qd; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 47.
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

[Running in non-interactive mode, continuing...]
```

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.

Nodes	4: Queue $\times$ 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`.

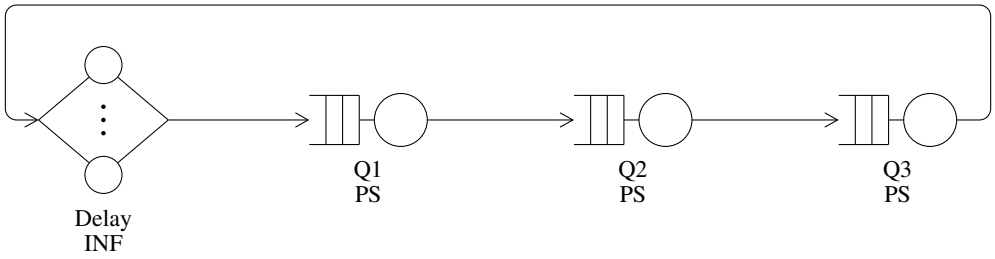

Figure 3.4: Topology of `ld_fes_multiclass`: a closed network over 4 queueing stations.

The example is built and solved as follows.

```
Network model = LoadDependentModel.ld_fes_multiclass();
System.out.println("MVA (original):");
new MVA(model, "method", "exact").getAvgTable().print();

FESResult fes = ModelAdapter.aggregateFES(
        model, subsetOf(model, new String[] {"Q1", "Q2", "Q3"}));
System.out.println("NC (FES model):");
new NC(fes.fesModel, "method", "exact").getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: java; 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
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.

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

NC analysis [method: conv; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

```
Network model = LoadDependentModel.ld_fes_singleclass();
System.out.println("MVA (original):");
```

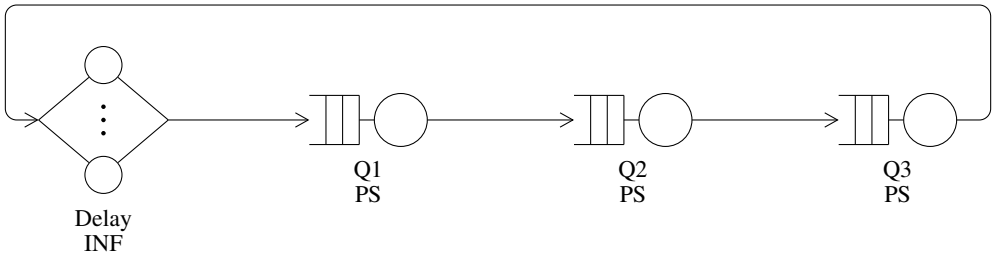

Figure 3.5: Topology of `ld_fes_singleclass`: a closed network over 4 queueing stations.

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`.

```
new MVA(model, "method", "exact").getAvgTable().print();

FESResult fes = ModelAdapter.aggregateFES(
        model, subsetOf(model, new String[] {"Q1", "Q2", "Q3"}));
System.out.println("NC (FES model):");
new NC(fes.fesModel, "method", "exact").getAvgTable().print();
pauseForUser();
```

Running this edition prints

```
MVA (original):
MVA analysis [method: exact; type: exact, deterministic; lang: java; 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
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
NC (FES model):
NC analysis [method: conv; type: exact, deterministic; lang: java; 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

[Running in non-interactive mode, continuing...]
```

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.

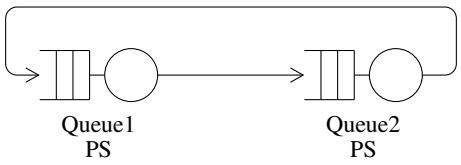

Figure 3.6: Topology of `ld_global_dependence`: a closed network over 2 queueing stations.

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.

```
int N = 3;

Network gdmodel = new Network("model");

Queue node1 = new Queue(gdmodel, "Queue1", SchedStrategy.PS);
Queue node2 = new Queue(gdmodel, "Queue2", SchedStrategy.PS);

ClosedClass jobclass1 = new ClosedClass(gdmodel, "Class1", N, node1, 0);
node1.setService(jobclass1, Exp.fitMean(1.0));
node2.setService(jobclass1, Exp.fitMean(0.5));

RoutingMatrix routingMatrix = gdmodel.initRoutingMatrix();
routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.0);
routingMatrix.set(jobclass1, jobclass1, node2, node1, 1.0);
gdmodel.link(routingMatrix);

final int M = gdmodel.getNumberOfStations();
final int K = gdmodel.getNumberOfClasses();
// phi as SolverCTMC sees it: an (nstations x nclasses) matrix of scalings
SerializableFunction<Matrix, Matrix> phi =
        n -> {
            Matrix v = Matrix.ones(M, K);
            double tot = 0;
            for (int i = 0; i < M; i++) {
                for (int r = 0; r < K; r++) tot += n.get(i, r);
            }

    ... (9 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

// peak scaling is 1: no station ever receives more than the whole link
Matrix gdPeak = new Matrix(1, 1);
gdPeak.set(0, 0, 1.0);
gdmodel.setGlobalDependence(phi, gdPeak);

return gdmodel;
```

Running this edition prints

```
jline.lang.Network@108c4c35
```

3.1.7 `ld_joint_dependence`

Joint dependence on the populations of several classes.
The example is built and solved as follows.

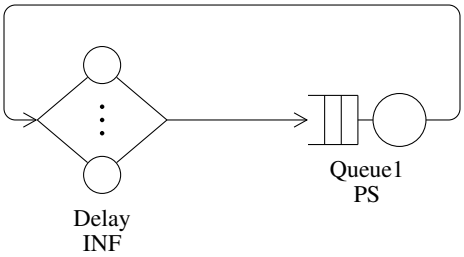

Figure 3.7: Topology of `ld_joint_dependence`: a closed network over 2 queueing stations.

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 3.7: Features of `ld_joint_dependence`.

```
Network model = LoadDependentModel.ld_joint_dependence();

// JMT is not solved here, as in ld_joint_dependence.m: the JSIM writer
// has no representation for the joint-dependence handle.
new CTMC(model, "exact").getAvgTable().print();
new MVA(model, "method", "qd").getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: exact; type: exact, deterministic; lang: java; 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 analysis [method: qd; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 47.
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

[Running in non-interactive mode, continuing...]
```

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.

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.

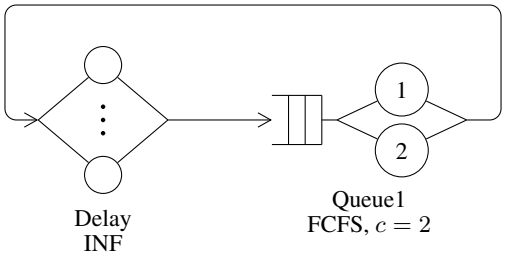

Figure 3.8: Topology of `ld_multiserver_fcfs`: a closed network over 2 queueing stations.

```
Network model = LoadDependentModel.ld_multiserver_fcfs();

// The reference's order on the load-dependent model: CTMC, then exact MVA,
// then JMT at seed 23000. Its golden holds the FIRST table each solver
// produced, so the order and the pinned method are part of the answer.
try {
    new CTMC(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new MVA(model, "method", "exact").getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new JMT(model, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
Queue1   Class1    14.667       1     11      11  1.3333  1.3333
MVA analysis [method: exact; type: exact, deterministic; lang: java; 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
Queue1   Class1    14.667       1     11      11  1.3333  1.3333
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass    QLen    Util    RespT   ResidT    ArvR    Tput
Delay    Class1    1.3535  1.3535  0.99717  0.99717  1.3436  1.3395
Queue1   Class1    14.647       1   10.962   10.962  1.3395  1.3397

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 3 solvers the script runs (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. The example is built and solved as follows.

```
Network model = LoadDependentModel.ld_multiserver_ps();

NetworkSolver[] solvers = new NetworkSolver[] {
    new NC(model, "method", "rd"),
    new MVA(model, "method", "exact"),
```

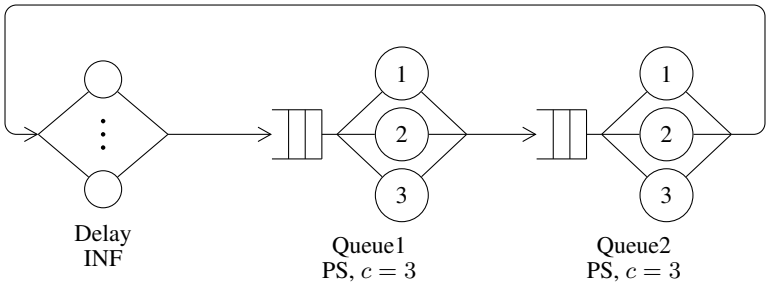

Figure 3.9: Topology of `ld_multiserver_ps`: a closed network over 3 queueing stations.

Nodes	3: Queue × 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`.

```
    new JMT(model, "seed", 12345)
};

for (NetworkSolver solver : solvers) {
    try {

        if (solver instanceof JMT) {
            // The solver's OWN options, not a fresh defaultOptions(): the latter
            // draws a RANDOM seed in its constructor (SolverOptions ->
            // RandomManager.generateRandomSeed), so replacing the object wholesale
            // discards the seed passed to the constructor above and makes this
            // example irreproducible -- which is what left the ld_multiserver_ps
            // parity row disagreeing with its golden on a Monte-Carlo margin.
            SolverOptions options = solver.getOptions();
            options.samples = 100000;
            ((JMT)solver).setOptions(options);
        }

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

Running this edition prints

```
NC analysis [method: rd; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.43985  0.43985       1       1  0.43985  0.43985
Delay    Class2    0.30095  0.30095       2       2  0.15047  0.15047
Queue1   Class1    0.78332  0.21993  1.7809  1.7809  0.43985  0.43985
Queue1   Class2     0.4437   0.1254  2.9486  2.9486  0.15047  0.15047
Queue2   Class1     2.7768  0.51316  6.3131  6.3131  0.43985  0.43985
Queue2   Class2     1.2554  0.22571  8.3426  8.3426  0.15047  0.15047
MVA analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
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

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

```
Delay    Class1    0.54287  0.54287   1.001   1.001  0.54723  0.54724
Delay    Class2    0.37154  0.37154  2.0053  2.0053   0.1845  0.18529
Queue1   Class1    0.85929   0.2743  1.5728  1.5728  0.54724  0.54724
Queue1   Class2    0.48185  0.15345  2.5991  2.5991  0.18529  0.18452
Queue2   Class1      2.597  0.64002  4.7503  4.7503  0.54724  0.54723
Queue2   Class2     1.1497  0.27421   6.184   6.184  0.18452  0.18514

[Running in non-interactive mode, continuing...]
```

The rows repeat for each of the 3 solvers the script runs (NC, MVA, 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.

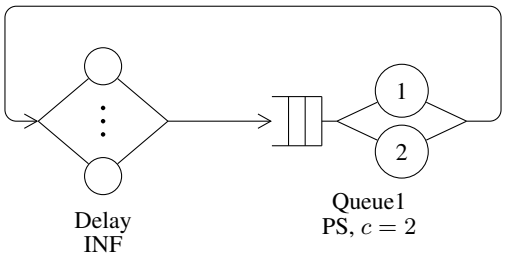

Figure 3.10: Topology of `ld_multiserver_ps_twoclasses`: a closed network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
Network model = LoadDependentModel.ld_multiserver_ps_twoclasses();

NetworkSolver[] solvers = new NetworkSolver[] {
    new NC(model, "method", "rd"),
    new MVA(model, "method", "exact"),
    new JMT(model, "seed", 12345)
};

for (NetworkSolver solver : solvers) {
    try {

        if (solver instanceof JMT) {
            // The solver's OWN options, not a fresh defaultOptions(): the latter
            // draws a RANDOM seed in its constructor (SolverOptions ->
            // RandomManager.generateRandomSeed), so replacing the object wholesale
            // discards the seed passed to the constructor above and makes this
            // example irreproducible -- which is what left the ld_multiserver_ps
            // parity row disagreeing with its golden on a Monte-Carlo margin.
            SolverOptions options = solver.getOptions();
            options.samples = 100000;
            ((JMT)solver).setOptions(options);
        }

        solver.getAvgTable().print();
    } catch (Exception e) {
```

```
    }
}

pauseForUser();
```

Running this edition prints

```
jline.lang.Network@736e9adb
```

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.

```
final int[][] A = {{1, 1, 0}, {1, 0, 1}};
final double[] C = {1.0, 1.0};
final double[] nu = {0.20, 0.30, 0.30};
final double[] mu = {1.00, 1.00, 1.00};
final int CUTOFF = 3;
final int S = 3;
final int base = CUTOFF + 1;
final double[] Phi = bfBalanceFunction(A, C, CUTOFF);

Network model = new Network("model");
Source source = new Source(model, "Source");
Queue[] routes = new Queue[S];
for (int s = 0; s < S; s++) {
    routes[s] = new Queue(model, "Route" + (s + 1), SchedStrategy.PS);
}
Sink sink = new Sink(model, "Sink");
OpenClass[] classes = new OpenClass[S];
for (int s = 0; s < S; s++) {
    classes[s] = new OpenClass(model, "Route" + (s + 1) + "Flows");
    source.setArrival(classes[s], new Exp(nu[s]));
}
for (int s = 0; s < S; s++) {
    for (int t = 0; t < S; t++) {
        if (s == t) {
            routes[t].setService(classes[s], new Exp(mu[s]));
        } else {

    ... (33 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        }
    }
    return v;
}, Matrix.ones(1, 1));

return model;
```

Running this edition prints

```
jline.lang.Network@3c5a99da
```

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
Network model1 = new Network("FCR MM1K Dropping");
Source source = new Source(model1, "Source");
Queue queue = new Queue(model1, "Queue", SchedStrategy.FCFS);
Sink sink = new Sink(model1, "Sink");
OpenClass jobclass = new OpenClass(model1, "Class1", 0);
source.setArrival(jobclass, Exp.fitRate(0.8));
queue.setService(jobclass, Exp.fitRate(1.0));
model1.link(Network.serialRouting(source, queue, sink));

model1.addRegion(Arrays.asList(queue));
Region fcr = model1.getRegions().get(0);
fcr.setGlobalMaxJobs(K);
fcr.setDropRule(jobclass, true);    // true = drop, false = wait outside
```

3.2.1 `fcr_constraints`

Per-class capacities inside one region, and the interaction with the global limit.

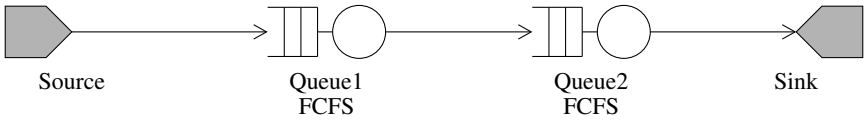

Figure 3.11: Topology of `fcr_constraints`: an open network over 2 queueing stations.

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.

```
System.out.println("=== FCR Constraint Types Demo ===\n");
System.out.println("Region covers: Queue1, Queue2");
System.out.println("Global max jobs: 2");
System.out.println("HighPriority max: 2 (dropping)");
System.out.println("LowPriority max: 2 (dropping)\n");

Network model = FCRegionModel.fcr_constraints();
JMT solver = new JMT(model, "seed", 23000, "samples", 100000, "verbose", VerboseLevel.SILENT)
    ;
NetworkAvgTable avgTable = solver.getAvgTable();

System.out.println(avgTable);
```

```
System.out.println("\nNote: jobs of either class are dropped once the region reaches");
System.out.println("its global limit or that class's own limit.");
```

Running this edition prints

```
=== FCR Constraint Types Demo ===

Region covers: Queue1, Queue2
Global max jobs: 2
HighPriority max: 2 (dropping)
LowPriority max: 2 (dropping)

WARNING: [refreshStruct] 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: [refreshStruct] Message casted more than once, repetitions will not be printed on screen for 60
     seconds.
jline.solvers.NetworkAvgTable@3ec300f1

Note: jobs of either class are dropped once the region reaches
its global limit or that class's own limit.
```

3.2.2 `fcr_lincon`

A linear constraint over the class populations, the general form of a region limit.

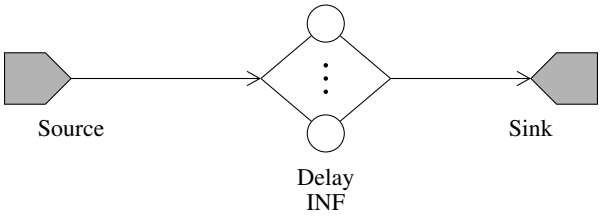

Figure 3.12: Topology of `fcr_lincon`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
System.out.println("=== FCR Linear Admission Constraints (LDES) ===\n");

double[][] cases_A = {
    // Case 1: Per-class only (A = I)
    {1.0, 0.0, 0.0, 1.0},
    // Case 2: Global + per-class
    {1.0, 0.0, 0.0, 1.0, 1.0, 1.0},
    // Case 3: General weighted
    {2.0, 1.0, 1.0, 3.0}
};
int[] cases_C = {2, 3, 2};
double[][] cases_b = {
    {3.0, 3.0},
    {3.0, 3.0, 5.0},
    {5.0, 7.0}
};
String[] caseNames = {
```

```
    "Case 1: Per-class only (A=I)",
    "Case 2: Global + per-class",
    "Case 3: General weighted"
};

for (int c = 0; c < 3; c++) {
    System.out.println("\n--- " + caseNames[c] + " ---");
    int C = cases_C[c];
    int K = 2;

    ... (8 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    Network model = FCRegionModel.fcr_lincon(A, b);

    SolverLDES solver = new SolverLDES(model, "seed", 23000, "samples", 500000, "verbose",
    VerboseLevel.SILENT);
    NetworkAvgTable avgTable = solver.getAvgTable();
    System.out.println(avgTable);
}
```

Running this edition prints

```
=== FCR Linear Admission Constraints (LDES) ===

--- Case 1: Per-class only (A=I) ---
WARNING: [refreshStruct] 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: [refreshStruct] Message casted more than once, repetitions will not be printed on screen for 60
     seconds.
jline.solvers.NetworkAvgTable@2a70a3d8

--- Case 2: Global + per-class ---
jline.solvers.NetworkAvgTable@2de8284b

--- Case 3: General weighted ---
jline.solvers.NetworkAvgTable@18bf3d14
```

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.

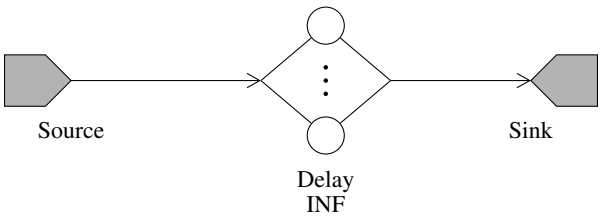

Figure 3.13: Topology of `fcr_lossn`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
System.out.println("=== FCR Loss Network (NC Solver) ===\n");
System.out.println("Model: Single Delay node in FCR with DROP policy");
System.out.println("Global max: 5, Class1 max: 3, Class2 max: 3\n");

Network model = FCRegionModel.fcr_lossn();

// Run NC solver (uses lossn method)
System.out.println("Running NC solver (lossn method)...");
SolverNC solverNC = new SolverNC(model, "verbose", VerboseLevel.SILENT);
NetworkAvgTable avgTableNC = solverNC.getAvgTable();
System.out.println("\nNC Results:");
System.out.println(avgTableNC);

// Run JMT for comparison
System.out.println("\nRunning JMT for comparison...");
JMT solverJMT = new JMT(model, "seed", 23000, "samples", 500000, "verbose", VerboseLevel.
    SILENT);
NetworkAvgTable avgTableJMT = solverJMT.getAvgTable();
System.out.println("\nJMT Results:");
System.out.println(avgTableJMT);

// Compare throughputs at the Delay node; the NC and JMT tables list
// different station sets, so rows are located by station/class name
System.out.println("\n=== Comparison (Delay node) ===");
System.out.printf("%-15s %12s %12s %12s%n", "Class", "NC Tput", "JMT Tput", "Rel Err");
for (int r = 0; r < 2; r++) {
    String className = "Class" + (r + 1);

    ... (3 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    System.out.printf("%-15s %12.4f %12.4f %10.2f%%%n",
            className, tputNC, tputJMT, relErr);
}

System.out.println("\nNote: NC uses analytical Erlang fixed-point approximation.");
System.out.println("JMT uses discrete-event simulation.");
```

Running this edition prints

```
=== FCR Loss Network (NC Solver) ===

Model: Single Delay node in FCR with DROP policy
Global max: 5, Class1 max: 3, Class2 max: 3

Running NC solver (lossn method)...
WARNING: [refreshStruct] 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: [refreshStruct] Message casted more than once, repetitions will not be printed on screen for 60
     seconds.

NC Results:
jline.solvers.NetworkAvgTable@3a883ce7

Running JMT for comparison...

JMT Results:
jline.solvers.NetworkAvgTable@6f3b5d16

=== Comparison (Delay node) ===
Class                NC Tput      JMT Tput      Rel Err
Class1                0.2990        0.2990        0.01%
Class2                0.1996        0.1998        0.12%

Note: NC uses analytical Erlang fixed-point approximation.
JMT uses discrete-event simulation.
```

3.2.4 `fcr_mm1kdrop`

A region of capacity K around one queue with dropping, which reproduces an M/M/1/K queue.

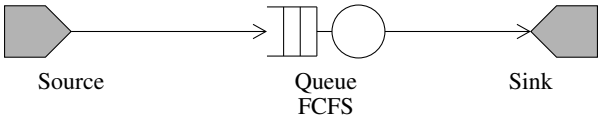

Figure 3.14: Topology of `fcr_mm1kdrop`: an open network over 1 queueing station.

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.

```
int K = 3;
System.out.printf("=== Comparison: FCR Dropping vs M/M/1/K (K=%d) ===%n%n", K);

// FCR dropping model
Network model1 = FCRegionModel.fcr_mm1kdrop();
JMT solver1 = new JMT(model1, "seed", 23000, "samples", 100000, "verbose", VerboseLevel.
    SILENT);
NetworkAvgTable avgTable1 = solver1.getAvgTable();

// Standard M/M/1/K
Network model2 = FCRegionModel.mm1k();
JMT solver2 = new JMT(model2, "seed", 23000, "samples", 100000, "verbose", VerboseLevel.
    SILENT);
NetworkAvgTable avgTable2 = solver2.getAvgTable();

// Compare results (queue is at index 1)
int queueIdx = 1;
System.out.println("Queue Metrics:");
System.out.printf("                    FCR Dropping    M/M/1/K%n");
System.out.printf("Queue Length:       %.4f          %.4f%n", avgTable1.getQLen().get(
    queueIdx), avgTable2.getQLen().get(queueIdx));
System.out.printf("Utilization:        %.4f          %.4f%n", avgTable1.getUtil().get(
    queueIdx), avgTable2.getUtil().get(queueIdx));
System.out.printf("Response Time:      %.4f          %.4f%n", avgTable1.getRespT().get(
    queueIdx), avgTable2.getRespT().get(queueIdx));
System.out.printf("Throughput:         %.4f          %.4f%n", avgTable1.getTput().get(
    queueIdx), avgTable2.getTput().get(queueIdx));

// Theoretical M/M/1/K values
double arrivalRate = 0.8;
double serviceRate = 1.0;
double rho = arrivalRate / serviceRate;

    ... (10 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

System.out.printf("Utilization:        %.4f%n", theoretical_U);
System.out.printf("Response Time:      %.4f%n", theoretical_R);
System.out.printf("Throughput:         %.4f%n", theoretical_X);
System.out.printf("Blocking Prob:      %.4f%n", pK);

System.out.println("\n==> FCR with dropping matches M/M/1/K (jobs lost when full)");
```

Running this edition prints

```
=== Comparison: FCR Dropping vs M/M/1/K (K=3) ===

Queue Metrics:
                    FCR Dropping    M/M/1/K
Queue Length:       1.2175          1.2212
```

```
Utilization:        0.6612          0.6612
Response Time:      1.8344          1.8344
Throughput:         0.6621          0.6621

Theoretical M/M/1/K:
Queue Length:       1.2249
Utilization:        0.6612
Response Time:      1.8525
Throughput:         0.6612
Blocking Prob:      0.1734

==> FCR with dropping matches M/M/1/K (jobs lost when full)
```

3.2.5 `fcr_mm1waitq`

The same region with waiting instead of dropping, so blocked jobs queue outside the region.

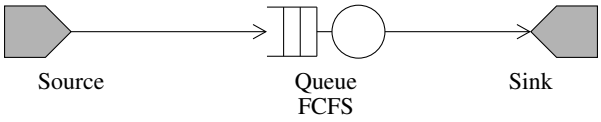

Figure 3.15: Topology of `fcr_mm1waitq`: an open network over 1 queueing station.

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.

```
System.out.println("=== Comparison: FCR Blocking vs M/M/1 ===\n");

// FCR blocking model
Network model1 = FCRegionModel.fcr_mm1waitq();
JMT solver1 = new JMT(model1, "seed", 23000, "samples", 100000, "verbose", VerboseLevel.
    SILENT);
// THE NODE TABLE, as the reference prints: its golden holds the FCR1 and
// Sink rows, which no station table carries.
solver1.getAvgNodeTable().print();
NetworkAvgTable avgTable1 = solver1.getAvgTable();

// Standard M/M/1
Network model2 = FCRegionModel.mm1();
JMT solver2 = new JMT(model2, "seed", 23000, "samples", 100000, "verbose", VerboseLevel.
    SILENT);
solver2.getAvgNodeTable().print();
NetworkAvgTable avgTable2 = solver2.getAvgTable();

// Compare results (queue is at index 1)
int queueIdx = 1;
System.out.println("Queue Metrics:");
System.out.printf("                    FCR Blocking    M/M/1%n");
System.out.printf("Queue Length:       %.4f          %.4f%n", avgTable1.getQLen().get(
    queueIdx), avgTable2.getQLen().get(queueIdx));
System.out.printf("Utilization:        %.4f          %.4f%n", avgTable1.getUtil().get(
    queueIdx), avgTable2.getUtil().get(queueIdx));
System.out.printf("Response Time:      %.4f          %.4f%n", avgTable1.getRespT().get(
    queueIdx), avgTable2.getRespT().get(queueIdx));
System.out.printf("Throughput:         %.4f          %.4f%n", avgTable1.getTput().get(
    queueIdx), avgTable2.getTput().get(queueIdx));
```

```
// Theoretical M/M/1 values

    ... (9 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

System.out.printf("Queue Length:       %.4f%n", theoretical_Q);
System.out.printf("Utilization:        %.4f%n", theoretical_U);
System.out.printf("Response Time:      %.4f%n", theoretical_R);
System.out.printf("Throughput:         %.4f%n", theoretical_X);

System.out.println("\n==> FCR with blocking matches M/M/1 (jobs wait, no loss)");
```

Running this edition prints

```
=== Comparison: FCR Blocking vs M/M/1 ===

Queue Metrics:
                    FCR Blocking    M/M/1
Queue Length:       0.9784          0.9786
Utilization:        0.4939          0.4939
Response Time:      1.9641          1.9643
Throughput:         0.4992          0.4992

Theoretical M/M/1:
Queue Length:       1.0000
Utilization:        0.5000
Response Time:      2.0000
Throughput:         0.5000

==> FCR with blocking matches M/M/1 (jobs wait, no loss)
```

3.2.6 `fcr_oqndrop`

A region spanning several stations of an open network, with dropping.

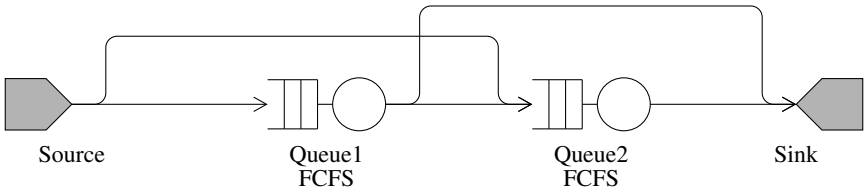

Figure 3.16: Topology of `fcr_oqndrop`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
System.out.println("=== FCR Multiclass Dropping Example ===\n");

Network model = FCRegionModel.fcr_oqndrop();
JMT solver = new JMT(model, "seed", 23000, "samples", 50000, "verbose", VerboseLevel.SILENT);
NetworkAvgTable avgTable = solver.getAvgTable();

System.out.println(avgTable);
```

Running this edition prints

```
=== FCR Multiclass Dropping Example ===

WARNING: [refreshStruct] 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: [refreshStruct] Message casted more than once, repetitions will not be printed on screen for 60
     seconds.
jline.solvers.NetworkAvgTable@3ec300f1
```

3.2.7 `fcr_oqnwaitq`

The same region with waiting.

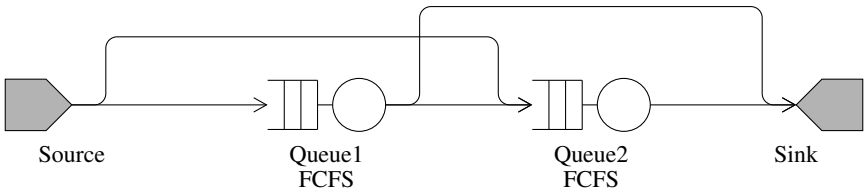

Figure 3.17: Topology of `fcr_oqnwaitq`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
System.out.println("=== FCR Multiclass Blocking Example ===\n");

Network model = FCRegionModel.fcr_oqnwaitq();
JMT solver = new JMT(model, "seed", 23000, "samples", 10000, "verbose", VerboseLevel.SILENT);
NetworkAvgTable avgTable = solver.getAvgTable();

System.out.println(avgTable);
```

Running this edition prints

```
=== FCR Multiclass Blocking Example ===

WARNING: [refreshStruct] 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: [refreshStruct] Message casted more than once, repetitions will not be printed on screen for 60
     seconds.
jline.solvers.NetworkAvgTable@3ec300f1
```

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`.

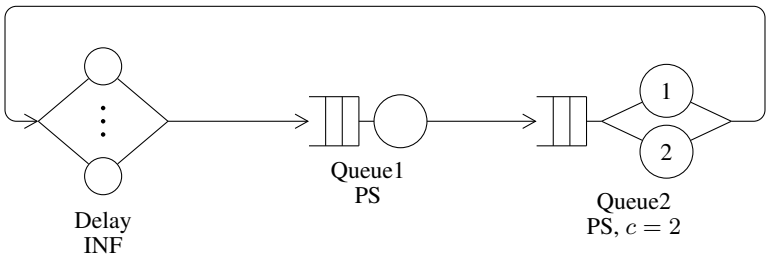

Figure 3.18: Topology of `statepr_aggr`: a closed network over 3 queueing stations.

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.

```
Network model = StateProbabilitiesModel.statepr_aggr();

NetworkSolver solver = new CTMC(model);

try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass  QLen  Util    RespT   ResidT  ArvR  Tput
Delay    Class1    0.84  0.84        1        1  0.84  0.84
Queue1   Class1    0.32  0.28  0.38095  0.38095  0.84  0.84
Queue2   Class1    0.84  0.42        1        1  0.84  0.84

[Running in non-interactive mode, continuing...]
```

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`.

3.3.2 statepr_aggr_large

The same on a model whose state space is large enough to need the aggregate form.
The example is built and solved as follows.

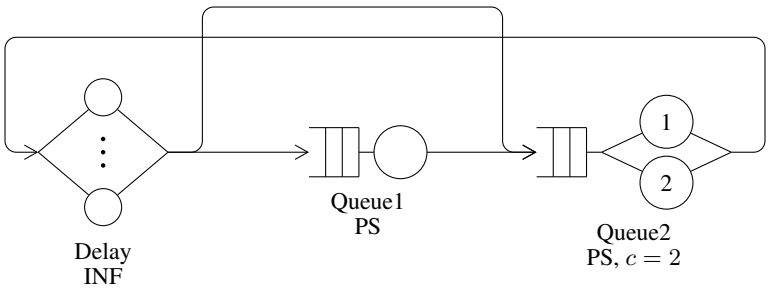

Figure 3.19: Topology of `statepr_aggr_large`: a closed network over 3 queueing stations.

Nodes	3: Queue × 2, Delay
Classes	4: Class1 (closed, $N = 1$), Class2 (closed, $N = 0$), Class3 (closed, $N = 4$), 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`.

```
Network model = StateProbabilitiesModel.statepr_aggr_large();

NetworkSolver solver = new JMT(model, "seed", 12345);

try {
    // The solver's OWN options, not a fresh defaultOptions(): the latter
    // draws a RANDOM seed in its constructor (SolverOptions ->
    // RandomManager.generateRandomSeed), so replacing the object wholesale
    // discards the seed passed to the constructor above and makes this
    // example irreproducible -- which is what left the ld_multiserver_ps
    // parity row disagreeing with its golden on a Monte-Carlo margin.
    SolverOptions options = solver.getOptions();
    options.samples = 100000;
    ((JMT)solver).setOptions(options);

    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT    ResidT     ArvR     Tput
Delay    Class1     0.25967   0.25967    1.0024    0.5012  0.26031  0.26021
Delay    Class2     0.13087   0.13087   0.50138   0.25069  0.26062  0.26033
Delay    Class3      1.2691    1.2691   0.99802   0.49901   1.2699   1.2698
Delay    Class4      1.2758    1.2758    1.0048    0.5024   1.2699   1.2699
Queue1   Class1     0.11622   0.08639   0.44491   0.22245  0.26021  0.26021
Queue1   Class2    0.085733  0.064238   0.33284   0.16642  0.26033  0.26061
Queue1   Class3     0.37356   0.25339    0.2901   0.14505   1.2698   1.2698
Queue2   Class1     0.30435    0.1323     1.167   0.58348  0.26021  0.26062
Queue2   Class2     0.10126  0.043418    0.3899   0.19495  0.26061  0.26033
Queue2   Class3     0.30568   0.12745   0.24437   0.12218   1.2698   1.2699
Queue2   Class4     0.77755   0.31689   0.61207   0.30603   1.2699   1.2699

[Running in non-interactive mode, continuing...]
```

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.

3.3.3 statepr_allprobs_fcfs

The FCFS version, where the detailed state carries the order of the jobs in service.

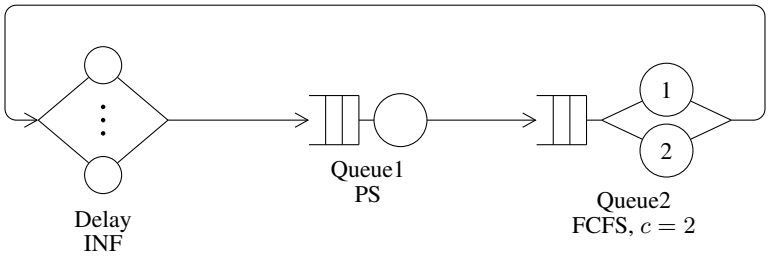

Figure 3.20: Topology of statepr_allprobs_fcfs: a closed network over 3 queueing stations.

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.

The example is built and solved as follows.

```
Network model = StateProbabilitiesModel.statepr_allprobs_fcfs();

NetworkSolver solver = new CTMC(model);

try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util     RespT    ResidT    ArvR     Tput
Delay    Class1     1.1538   1.1538         1         1  1.1538  1.1538
Queue1   Class1    0.46154  0.38462       0.4       0.4  1.1538  1.1538
Queue2   Class2    0.38462  0.19231   0.33333   0.33333  1.1538  1.1538

[Running in non-interactive mode, continuing...]
```

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.

3.3.4 statepr_allprobs_ps

Every probability API on one PS model: detailed and aggregate, per station and system-wide.
The example is built and solved as follows.

```
Network model = StateProbabilitiesModel.statepr_allprobs_ps();

NetworkSolver solver = new CTMC(model);
```

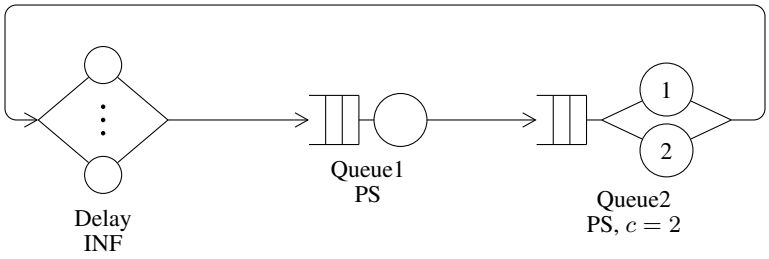

Figure 3.21: Topology of `statepr_allprobs_ps`: a closed network over 3 queueing stations.

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, SSA, JMT

Table 3.21: Features of `statepr_allprobs_ps`.

```
try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT    ArvR    Tput
Delay    Class1     1.1538   1.1538        1        1  1.1538  1.1538
Queue1   Class1    0.46154  0.38462      0.4      0.4  1.1538  1.1538
Queue2   Class2    0.38462  0.19231  0.33333  0.33333  1.1538  1.1538

[Running in non-interactive mode, continuing...]
```

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`.

3.3.5 `statepr_sys_aggr`

The joint aggregate state of the whole network, through `getProbSysAggr`.

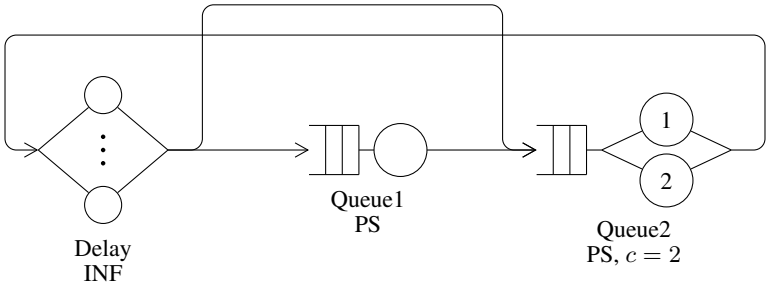

Figure 3.22: Topology of `statepr_sys_aggr`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
Network model = StateProbabilitiesModel.statepr_sys_aggr();
```

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`.

```
NetworkSolver solver = new CTMC(model);

try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass      QLen      Util     RespT   ResidT     ArvR     Tput
Delay    Class1     0.27272   0.27272         1      0.5  0.27272  0.27272
Delay    Class2     0.13636   0.13636       0.5     0.25  0.27272  0.27272
Delay    Class3     0.98217   0.98217         1      0.5  0.98217  0.98217
Delay    Class4     0.98217   0.98217         1      0.5  0.98217  0.98217
Queue1   Class1     0.11211  0.090907   0.41109  0.20554  0.27272  0.27272
Queue1   Class2    0.084084   0.06818   0.30832  0.15416  0.27272  0.27272
Queue1   Class3     0.26672   0.19643   0.27156  0.13578  0.98217  0.98217
Queue2   Class1     0.29604   0.13636    1.0855  0.54276  0.27272  0.27272
Queue2   Class2    0.098681  0.045453   0.36184  0.18092  0.27272  0.27272
Queue2   Class3      0.2197  0.098217   0.22369  0.11184  0.98217  0.98217
Queue2   Class4     0.54925   0.24554   0.55922  0.27961  0.98217  0.98217

[Running in non-interactive mode, continuing...]
```

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`.

3.3.6 `statepr_sys_aggr_large`

The large-model version.

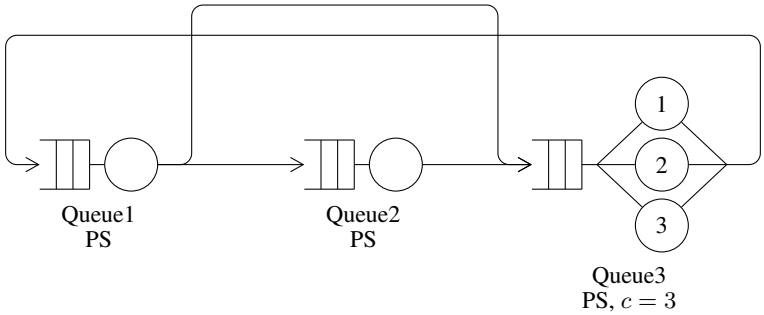

Figure 3.23: Topology of `statepr_sys_aggr_large`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
Network model = StateProbabilitiesModel.statepr_sys_aggr_large();
```

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`.

```
NetworkSolver solver = new JMT(model, "seed", 12345);

try {
    // The solver's OWN options, not a fresh defaultOptions(): the latter
    // draws a RANDOM seed in its constructor (SolverOptions ->
    // RandomManager.generateRandomSeed), so replacing the object wholesale
    // discards the seed passed to the constructor above and makes this
    // example irreproducible -- which is what left the ld_multiserver_ps
    // parity row disagreeing with its golden on a Monte-Carlo margin.
    SolverOptions options = solver.getOptions();
    options.samples = 100000;
    ((JMT)solver).setOptions(options);

    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
jline.lang.Network@6438a396
```

3.3.7 `statepr_sys_marg`

The joint law of the per-station TOTAL queue lengths, all classes summed out, through `getProbSysMarg`.

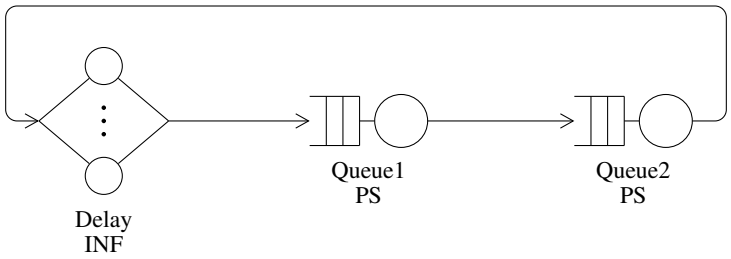

Figure 3.24: Topology of `statepr_sys_marg`: a closed network over 3 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

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.

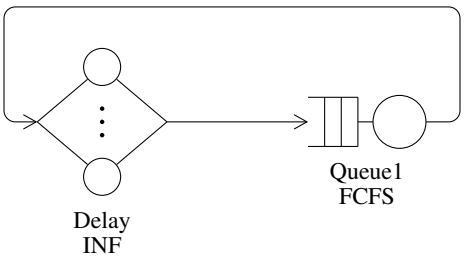

Figure 3.25: Topology of `init_state_fcfs_exp`: a closed network over 2 queueing stations.

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.

```
Network model = InitStateModel.init_state_fcfs_exp();

try {
    new CTMC(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new FLD(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("FLD failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
=== Example 1: Two Initialization Methods ===

Method 1: Default initialization
Initial state: All jobs start at reference station
Fluid solver completed for default initialization

Method 2: Initialize from marginal distribution [2;3]
Initial state: 2 jobs at station 1, 3 jobs at station 2
Fluid solver completed for marginal initialization
Example 1 completed: Different initial states affect transient behavior
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.69951  0.69951       1       1  0.69951  0.69951
Queue1   Class1     4.3005   0.9993  6.1478  6.1478  0.69951  0.69951
Fluid analysis [method: minnormal; type: approximate, deterministic; lang: java; env: python] completed in
      <t>s.
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.69999  0.69999       1       1  0.69999  0.69999
Queue1   Class1        4.3  0.99999  6.1429  6.1429  0.69999  0.69999

[Running in non-interactive mode, continuing...]
```

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.

3.4.2 `init_state_fcfs_nonexp`

The same with non-exponential service, where the phase of the job in service is part of the state.

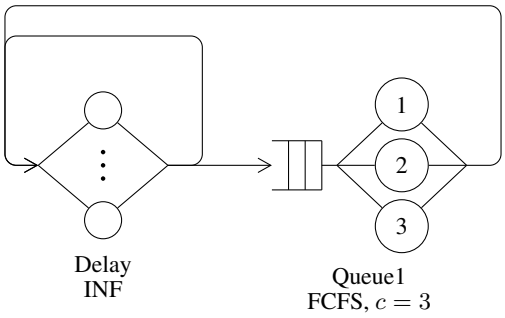

Figure 3.26: Topology of `init_state_fcfs_nonexp`: a closed network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
Network model = InitStateModel.init_state_fcfs_nonexp();

try {
    new CTMC(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new FLD(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
```

```
    System.out.println("FLD failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
=== Example 2: Three Initialization Approaches ===

Method 1: Default initialization
Initial state: All jobs start at reference station
Fluid solver completed for default initialization

Method 2: Initialize from marginal distribution [0,0;4,1]
Initial state: 4 jobs of class 1 and 1 job of class 2 at station 2
Fluid solver completed for marginal initialization

Method 3: Uniform prior over states with same job counts
Initial state: Uniform distribution over all valid states with same job counts
Fluid solver completed for uniform prior initialization
Example 2 completed: Different initialization approaches affect transient behavior
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen      Util    RespT   ResidT     ArvR     Tput

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

Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
Station  JobClass     QLen      Util    RespT   ResidT     ArvR     Tput
Delay    Class1     2.0273    2.0273        1   1.2821   2.0273   2.0273
Delay    Class2     1.3786    1.3786        1  0.87179   1.3786   1.3786
Queue1   Class1    0.17404  0.056313  0.85847  0.11006  0.20273  0.20273
Queue1   Class2     1.4201   0.45952   1.0302  0.89809   1.3786   1.3786

[Running in non-interactive mode, continuing...]
```

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.

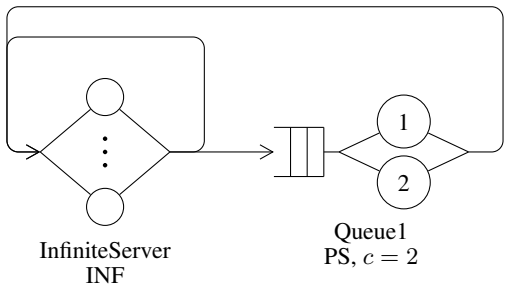

Figure 3.27: Topology of `init_state_ps`: a closed network over 2 queueing stations.

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.

```
Network model = InitStateModel.init_state_ps();

try {
    new CTMC(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new JMT(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
try {
    new SSA(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("SSA failed: " + e.getMessage());
}
try {
    new FLD(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("FLD failed: " + e.getMessage());
}
try {
    new MVA(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new NC(model, "verbose", 1, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("NC failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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
Queue1          Class2    0.92708  0.34326   1.3504  0.54659  0.68652  0.68652
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station         JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
InfiniteServer  Class1      0.332    0.332  0.33224  0.19776  0.99645  0.99646
InfiniteServer  Class2     1.3674   1.3674   1.9991  0.80918  0.68249  0.68251
Queue1          Class1     1.3654  0.51204   13.573  0.80794  0.10041  0.10043
Queue1          Class2    0.91601  0.34077   1.3463  0.54493  0.68251  0.67959
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
    >s. Iterations: 100000.
Station         JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
InfiniteServer  Class1      0.339    0.339  0.33333  0.19841   1.0077    1.017

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

NC analysis [method: default/exact/gld; type: approximate, deterministic; lang: java; 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
Queue1          Class2    0.92708  0.34326   1.3504  0.54659  0.68652  0.68652

[Running in non-interactive mode, continuing...]
```

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.

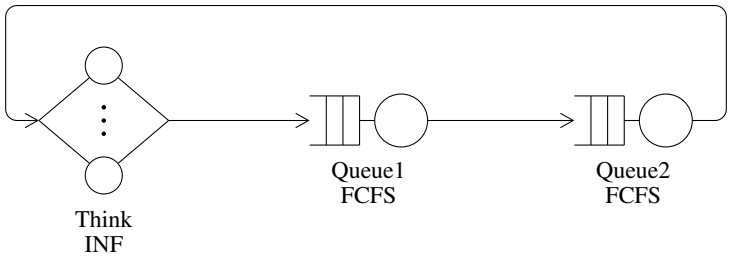

Figure 3.28: Topology of `ldes_warmstart`: a closed network over 3 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

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.

```
Network model = StateDepRoutingModel.sdroute_closed();

try {
```

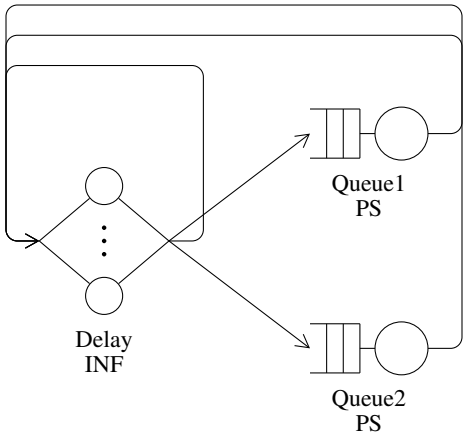

Figure 3.29: Topology of `sdroute_closed`: a closed network over 3 queueing stations.

```
    new CTMC(model, "keep", true).getAvgTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}
try {
    new JMT(model, "samples", 100000, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
try {
    new SSA(model, "verbose", true, "samples", 10000, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("SSA failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
CTMC analysis [method: default; type: exact, deterministic; lang: java; 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 Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Delay    Class1     0.6634   0.6634  0.99642  0.99642  0.67275  0.66595
Queue1   Class1     0.2227   0.2227  0.99408  0.33136  0.22136  0.22132
Queue2   Class1    0.11295  0.11295  0.50054  0.16685  0.22136  0.22136
SSA analysis [method: default/nrm; type: approximate, randomized; lang: java; env: python] completed in <t
    >s. Iterations: 10000.
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

[Running in non-interactive mode, continuing...]
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

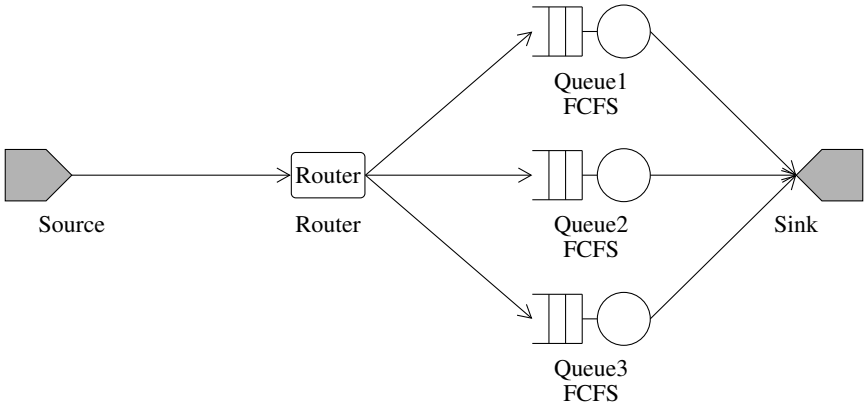

Figure 3.30: Topology of `sdroute_jsq`: an open network over 3 queueing stations with a router.

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`.

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

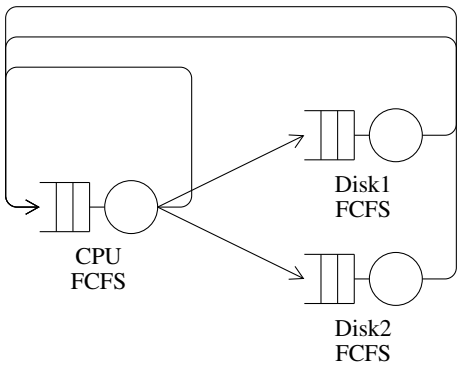

Figure 3.31: Topology of `sdroute_krzesinski`: a closed network over 3 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.5.5 `sdroute_open`

Round-robin dispatching from a Router node in an open network.

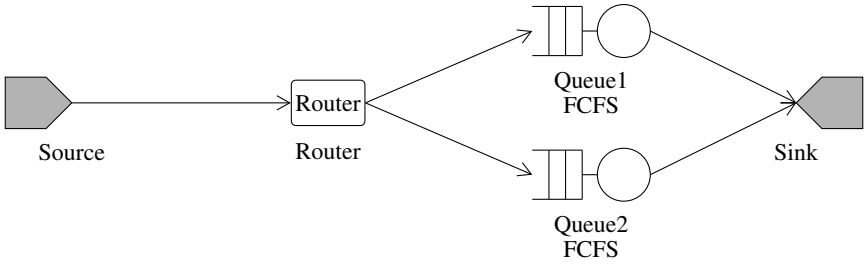

Figure 3.32: Topology of `sdroute_open`: an open network over 2 queueing stations with a router.

The example is built and solved as follows.

```
Network model = StateDepRoutingModel.sdroute_open();

try {
    new JMT(model, "seed", 23000).getAvgNodeTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
try {
    new CTMC(model, "cutoff", 5).getAvgNodeTable().print();
} catch (Exception e) {
    System.out.println("CTMC failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

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`.

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Node    JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source  Class1          0        0        0        0        0    1.012
Router  Class1          0        0        0        0    1.012    1.012
Queue1  Class1     0.3024  0.26218  0.57611  0.28806  0.50351  0.50533
Queue2  Class1    0.29317  0.25361  0.56951  0.28476  0.50531  0.50719
Sink    Class1          0        0        0        0    1.012        0
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 5 for open/mixed model. State space
     truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Node    JobClass     QLen     Util    RespT   ResidT     ArvR     Tput
Source  Class1          0        0        0        0        0  0.99844
Router  Class1          0        0        0        0  0.99844  0.99844
Queue1  Class1    0.28701  0.24961  0.57492  0.28746  0.49922  0.49922
Queue2  Class1    0.28701  0.24961  0.57492  0.28746  0.49922  0.49922
Sink    Class1          0        0        0        0  0.99844        0

[Running in non-interactive mode, continuing...]
```

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.

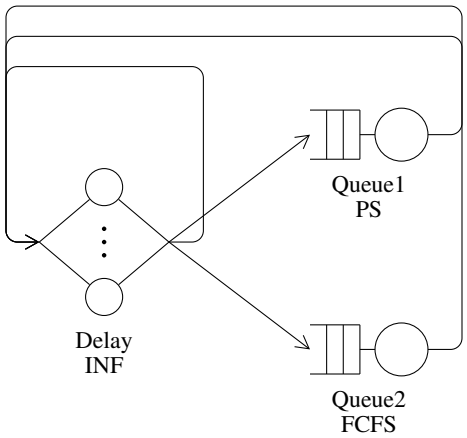

Figure 3.33: Topology of `sdroute_twoclasses_closed`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
Network model = StateDepRoutingModel.sdroute_twoclasses_closed();

// The reference's own seed and run length. They go through the
// constructor: setOptions(defaultOptions()) would REPLACE the seed with
// 0, which is drawn at random, and the row would differ run to run.
NetworkSolver solver = new JMT(model, "seed", 23000, "samples", 100000);
```

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`.

```
try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
jline.lang.Network@73035e27
```

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.

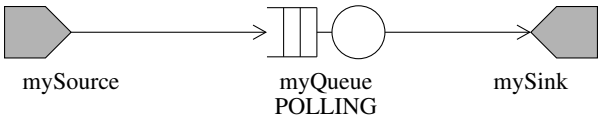

Figure 3.34: Topology of `polling_exhaustive_det`: an open network over 1 queueing station.

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.

```
Network model = CyclicPollingModel.polling_exhaustive_det();

// The reference runs MVA first and JMT second, at this seed and run length.
try {
    new MVA(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new JMT(model, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/stationtime; type: approximate, deterministic; lang: java; env: python]
    completed in <t>s.
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 Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
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  0.00099993  0.00099993  0.001   0.001     1     1
myQueue   myClass2   0.0019999  0.00099993  0.002   0.002     1     1

[Running in non-interactive mode, continuing...]
```

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.

3.6.2 `polling_exhaustive_exp`

Exhaustive service with exponential switchover times.

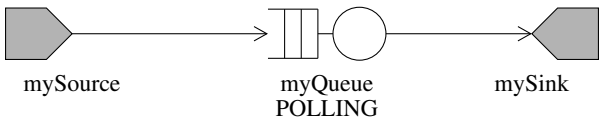

Figure 3.35: Topology of `polling_exhaustive_exp`: an open network over 1 queueing station.

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.

```
Network model = CyclicPollingModel.polling_exhaustive_exp();

// The reference runs MVA first and JMT second, at this seed and run length.
try {
```

```
    new MVA(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new JMT(model, "seed", 23000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/stationtime; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
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 Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station   JobClass      QLen      Util    RespT  ResidT      ArvR      Tput
mySource  myClass1         0         0        0       0         0  0.099696
mySource  myClass2         0         0        0       0         0   0.10064
myQueue   myClass1   0.11377  0.099441   1.1502  1.1502  0.099696  0.099232
myQueue   myClass2  0.082842  0.064638   0.8347  0.8347   0.10064   0.09965

[Running in non-interactive mode, continuing...]
```

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.

3.6.3 `polling_gated`

A gated polling server: at each visit it serves the jobs that were waiting when it arrived.

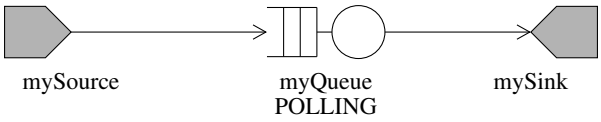

Figure 3.36: Topology of `polling_gated`: an open network over 1 queueing station.

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.

```
Network model = CyclicPollingModel.polling_gated();

// The reference runs MVA first and JMT second, at this seed and run length.
try {
    new MVA(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
```

```
    new JMT(model, "seed", 23000, "samples", 1000000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}

pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/stationtime; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
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 Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station   JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
mySource  myClass1       0        0        0       0        0   1.0012
mySource  myClass2       0        0        0       0        0  0.80026
myQueue   myClass1   11.31  0.24989    11.34   11.34   1.0012   1.0008
myQueue   myClass2  11.312  0.53337   14.149  14.149  0.80026  0.79995

[Running in non-interactive mode, continuing...]
```

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.

3.6.4 `polling_klimited`

k-limited service, at most k jobs per visit.

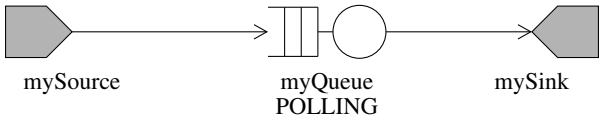

Figure 3.37: Topology of `polling_klimited`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Network model = CyclicPollingModel.polling_klimited();

// The reference runs MVA first and JMT second, at this seed and run length.
try {
    new MVA(model).getAvgTable().print();
} catch (Exception e) {
    System.out.println("MVA failed: " + e.getMessage());
}
try {
    new JMT(model, "seed", 23000, "samples", 100000).getAvgTable().print();
} catch (Exception e) {
    System.out.println("JMT failed: " + e.getMessage());
}
```

```
pauseForUser();
```

Running this edition prints

```
MVA analysis [method: default/stationtime; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
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 Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station    JobClass      QLen     Util    RespT  ResidT     ArvR     Tput
mySource   myClass1         0        0        0       0        0  0.19924
mySource   myClass2         0        0        0       0        0  0.30098
myQueue    myClass1   0.78887  0.19856   3.9795  3.9795  0.19924  0.20042
myQueue    myClass2    1.3909   0.2001   4.7392  4.7392  0.30098  0.30096

[Running in non-interactive mode, continuing...]
```

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.

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.

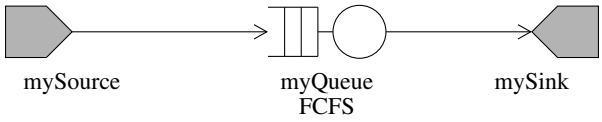

Figure 3.38: Topology of `switchover_basic`: an open network over 1 queueing station.

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.

```
Network model = SwitchoverTimesModel.switchover_basic();

// The reference's own seed and its own run length, which is the engine
// default: setOptions(defaultOptions()) would REPLACE the seed set here
// with 0, and a zero seed is drawn at random, so the row would not repeat.
NetworkSolver solver = new JMT(model, "seed", 23000, "keep", true);

try {
    solver.getAvgTable().print();
} catch (Exception e) {
```

```
}

pauseForUser();
```

Running this edition prints

```
jline.lang.Network@443b7951
```

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.

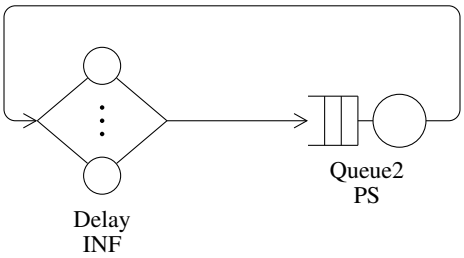

Figure 3.39: Topology of `cdf_respt_closed`: a closed network over 2 queueing stations.

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.

```
Network model = CDFRespTModel.cdf_respt_closed();

NetworkSolver[] solvers = new NetworkSolver[] {
    new JMT(model, "seed", 12345),
    new FLD(model)
};

for (NetworkSolver solver : solvers) {
    try {

        if (solver instanceof JMT) {
            JMT jmtSolver = (JMT) solver;
            // CDF analysis for JMT solver
            // Note: getCdfRespT() returns DistributionResult, not Matrix
            // Example shows general structure for CDF analysis
        } else if (solver instanceof FLD) {
            FLD fluidSolver = (FLD) solver;
            // CDF analysis for Fluid solver
```

```
            // Note: getCdfRespT() returns DistributionResult, not Matrix
            // Example shows general structure for CDF analysis
        }

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util     RespT    ResidT     ArvR     Tput
Delay    Class1    0.08793  0.08793  0.099239  0.099239  0.91269   0.9124
Queue2   Class1    0.91207  0.91207   0.99558   0.99558   0.9124  0.91224
Fluid analysis [method: minnormal; type: approximate, deterministic; lang: java; env: python] completed in
      <t>s.
Fluid analysis [method: default/minnormal; type: approximate, deterministic; lang: java; env: python]
     completed in <t>s.
Station  JobClass      QLen      Util  RespT  ResidT     ArvR     Tput
Delay    Class1    0.090909  0.090909    0.1     0.1  0.90909  0.90909
Queue2   Class1     0.90909   0.90909      1       1  0.90909  0.90909

[Running in non-interactive mode, continuing...]
```

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.

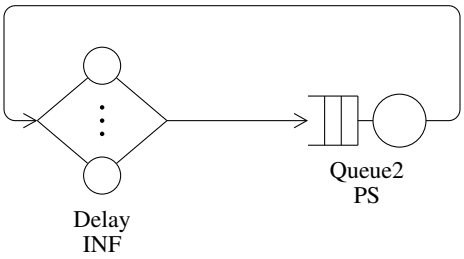

Figure 3.40: Topology of `cdf_respt_closed_threeclasses`: a closed network over 2 queueing stations.

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.

```
Network model = CDFRespTModel.cdf_respt_closed_threeclasses();

NetworkSolver[] solvers = new NetworkSolver[] {
    new JMT(model, "seed", 12345),
    new FLD(model)
```

```
};

for (NetworkSolver solver : solvers) {
    try {

        if (solver instanceof JMT) {
            JMT jmtSolver = (JMT) solver;
            // The solver's OWN options, not a fresh defaultOptions(): the latter
            // draws a RANDOM seed in its constructor (SolverOptions ->
            // RandomManager.generateRandomSeed), so replacing the object wholesale
            // discards the seed passed to the constructor above and makes this
            // example irreproducible -- which is what left the ld_multiserver_ps
            // parity row disagreeing with its golden on a Monte-Carlo margin.
            SolverOptions options = jmtSolver.getOptions();
            options.samples = 100000;
            jmtSolver.setOptions(options);
        }

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.8.3 cdf_respt_distrib

The distribution object returned by `getCdfRespT` and the percentiles read from it.

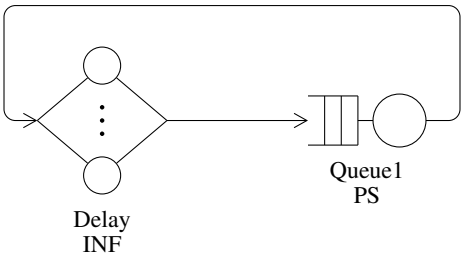

Figure 3.41: Topology of `cdf_respt_distrib`: a closed network over 2 queueing stations.

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.

```
Network model = CDFRespTModel.cdf_respt_distrib();

NetworkSolver[] solvers = new NetworkSolver[] {
    new JMT(model, "seed", 12345),
    new FLD(model)
};

for (NetworkSolver solver : solvers) {
```

```
    try {
        DistributionResult cdfResults = null;

        if (solver instanceof JMT) {
            JMT jmtSolver = (JMT) solver;
            SolverOptions options = JMT.defaultOptions();
            options.samples = 100000;
            options.seed = 23000;
            jmtSolver.setOptions(options);
            // Use getTranCdfRespT for transient analysis like MATLAB
            cdfResults = jmtSolver.getTranCdfRespT();
            System.out.println("JMT Solver Transient CDF Analysis (Distributions):");
        } else if (solver instanceof FLD) {
            FLD fluidSolver = (FLD) solver;
            cdfResults = fluidSolver.getCdfRespT();
            System.out.println("Fluid Solver CDF Analysis (Distributions):");
        }


    ... (4 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
JMT Solver Transient CDF Analysis (Distributions):

=== JMT CDF Statistical Analysis ===
Station 1, Class 1: Mean=1.0005, Variance=1.0064, SCV=1.0054
Station 1, Class 2: Mean=3.9898, Variance=8.0429, SCV=0.5052
Station 2, Class 1: Mean=6.4328, Variance=43.7579, SCV=1.0575
Station 2, Class 2: Mean=16.5309, Variance=7664.7234, SCV=28.0482

JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.13469  0.13469  1.0011  1.0011  0.13486  0.13486
Delay    Class2     0.5882   0.5882  3.9895  3.9895  0.14254  0.14592
Queue1   Class1    0.86531  0.26808  6.4659  6.4659  0.13486  0.13484

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


Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.13398  0.13398       1       1  0.13398  0.13398
Delay    Class2    0.58492  0.58492       4       4  0.14623  0.14623
Queue1   Class1    0.86602  0.26797  6.4637  6.4637  0.13398  0.13398
Queue1   Class2     2.4151  0.73115  16.516  16.516  0.14623  0.14623

[Running in non-interactive mode, continuing...]
```

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.

3.8.4 `cdf_respt_open_twoclasses`

The open two-class version, on FCFS stations.
The example is built and solved as follows.

```
Network model = CDFRespTModel.cdf_respt_open_twoclasses();

NetworkSolver[] solvers = new NetworkSolver[] {
    new JMT(model, "seed", 12345),
    new FLD(model)
```

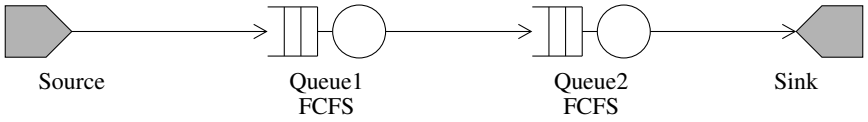

Figure 3.42: Topology of `cdf_respt_open_twoclasses`: an open network over 2 queueing stations.

Nodes	4: Source, Queue × 2, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue1: FCFS; Queue2: FCFS
Arrivals	Source – Class1: Exp(0.25); 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`.

```
};

for (NetworkSolver solver : solvers) {
    try {
        DistributionResult cdfResults = null;

        if (solver instanceof JMT) {
            JMT jmtSolver = (JMT) solver;
            SolverOptions options = JMT.defaultOptions();
            options.samples = 10000;
            options.seed = 23000;
            jmtSolver.setOptions(options);
            // Use getTranCdfRespT for transient analysis like MATLAB
            cdfResults = jmtSolver.getTranCdfRespT();
            System.out.println("JMT Solver Transient CDF Analysis (Open Two Classes):");
        } else if (solver instanceof FLD) {
            FLD fluidSolver = (FLD) solver;
            SolverOptions options = FLD.defaultOptions();
            options.iter_max = 300;
            fluidSolver.setOptions(options);
            cdfResults = fluidSolver.getCdfRespT();

    ... (7 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
JMT Solver Transient CDF Analysis (Open Two Classes):

=== JMT CDF Statistical Analysis ===
Station 2, Class 1: Mean=1.9357, Variance=3.7872, SCV=1.0107
Station 2, Class 2: Mean=1.9502, Variance=3.8886, SCV=1.0225
Station 3, Class 1: Mean=2.0904, Variance=4.4437, SCV=1.0169
Station 3, Class 2: Mean=2.0688, Variance=4.2944, SCV=1.0034

JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
Station  JobClass     QLen     Util    RespT    ResidT     ArvR     Tput
Source   Class1          0        0        0        0        0  0.24805
Source   Class2          0        0        0        0        0  0.25423
Queue1   Class1    0.48083  0.25248   1.9043  0.95215  0.24805  0.24804

... (14 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    0.37514  0.25  1.5005  0.75027  0.25  0.25
Queue1   Class2    0.37514  0.25  1.5005  0.75027  0.25  0.25
Queue2   Class1    0.37514  0.25  1.5005  0.75027  0.25  0.25
Queue2   Class2    0.37514  0.25  1.5005  0.75027  0.25  0.25

[Running in non-interactive mode, continuing...]
```

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.

3.8.5 cdf_respt_populations

How the response time distribution moves as the closed population grows.

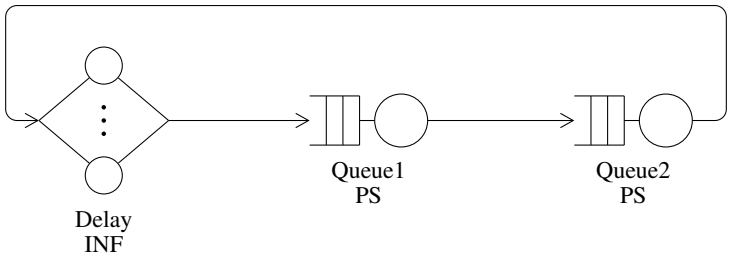

Figure 3.43: Topology of `cdf_respt_populations`: a closed network over 3 queueing stations.

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.

```
Network model = CDFRespTModel.cdf_respt_populationsN4();

NetworkSolver[] solvers = new NetworkSolver[] {
    new JMT(model, "seed", 12345),
    new FLD(model)
};

for (NetworkSolver solver : solvers) {
    try {
        DistributionResult cdfResults = null;

        if (solver instanceof JMT) {
            JMT jmtSolver = (JMT) solver;
            SolverOptions options = JMT.defaultOptions();
            options.seed = 12345;
            jmtSolver.setOptions(options);
            cdfResults = jmtSolver.getCdfRespT();
            System.out.println("JMT Solver CDF Analysis (Populations):");
        } else if (solver instanceof FLD) {
            FLD fluidSolver = (FLD) solver;
            SolverOptions options = FLD.defaultOptions();
            options.iter_max = 100;
            fluidSolver.setOptions(options);
            cdfResults = fluidSolver.getCdfRespT();
            System.out.println("Fluid Solver CDF Analysis (Populations):");
```

```
        }

    ... (5 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

        solver.getAvgTable().print();
    } catch (Exception e) {
    }
}

pauseForUser();
```

Running this edition prints

```
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
JMT Model: <tmp>
JMT analysis [method: default; type: approximate, randomized; lang: java; env: python] completed in <t>s.
JMT Solver CDF Analysis (Populations):

=== JMT CDF Statistical Analysis ===
Station 1, Class 1: Mean=1.0043, Variance=0.9943, SCV=0.9859
Station 2, Class 1: Mean=4.6754, Variance=26.0549, SCV=1.1919
Station 3, Class 1: Mean=4.6939, Variance=26.6422, SCV=1.2092

Station  JobClass     QLen     Util   RespT  ResidT     ArvR     Tput
Delay    Class1    0.38464  0.38464  1.0045  1.0045  0.38528  0.38519
Queue1   Class1     1.8326  0.77879  4.6705  4.6705  0.38519  0.38806
Queue2   Class1     1.7829  0.76221  4.6443  4.6443  0.38806  0.38539
Fluid analysis [method: dae; type: approximate, deterministic; lang: java; env: python] completed in <t>s.

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

Station 3, Class 1: Mean=4.2375, Variance=19.2421, SCV=1.0716

Station  JobClass    QLen     Util   RespT  ResidT    ArvR    Tput
Delay    Class1    0.3986   0.3986       1       1  0.3986  0.3986
Queue1   Class1    1.8007  0.79721  4.5175  4.5175  0.3986  0.3986
Queue2   Class1    1.8007  0.79721  4.5175  4.5175  0.3986  0.3986

[Running in non-interactive mode, continuing...]
```

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.

3.9 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.

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`.

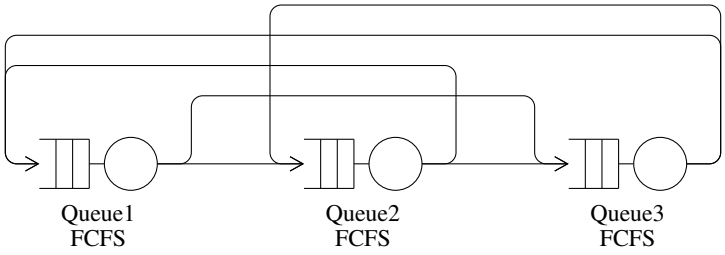

Figure 3.44: Topology of `busyp_subnetwork`: a closed network over 3 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

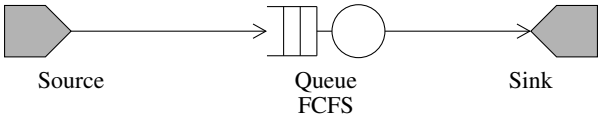

Figure 3.45: Topology of `snc_delay_quantile`: an open network over 1 queueing station.

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.

```
final double lambda = 0.6;
final double mu = 1.0;
Network model = new Network("SncQuantile");
Source source = new Source(model, "Source");
Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Sink");
OpenClass jobclass = new OpenClass(model, "Class1");
source.setArrival(jobclass, new Exp(lambda));
```

```
queue.setService(jobclass, new Exp(mu));
RoutingMatrix P = model.initRoutingMatrix();
P.set(jobclass, jobclass, Network.serialRouting(source, queue, sink));
model.link(P);

SolverBA solver = new SolverBA(model, "snc.upper");

// getPercTable is to the snc family what getAvgTable is to a mean
// solver: one row per station and class, reporting the response-time
// and queue-length quantiles at the requested violation probability.
NetworkPercTable percTable = solver.getPercTable(1e-3);
percTable.print();

// The exact M/M/1 sojourn tail is exp(-(mu-lambda)*d) and the exact
// queue-length tail is rho^(n+1). The bound reproduces both DECAY RATES
// exactly and pays a constant prefactor, so the ratio of the bounded
// quantile to the exact one falls towards 1 as eps is tightened: the
// family is at its best exactly where simulation is at its worst.

    ... (35 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

System.out.printf("     the optimal theta approaches log(mu/lambda) = %.4f, which is%n",
        Math.log(mu / lambda));
System.out.printf("     what makes the backlog decay rate exact%n");
SncResult back = Snc_bound_delay.snc_bound_delay(arv, srv, d.value);
System.out.printf("api: P{D > %.4f} <= %.3e (theta = %.4f), exact tail %.3e%n",
        d.value, back.value, back.theta, Math.exp(-(mu - lambda) * d.value));
```

Running this edition prints

```
Station  JobClass  RespTPerc  QLenPerc
Queue    Class1       29.608    23.645

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
BA analysis [method: snc.upper; type: upper bound, deterministic; lang: java; 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    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/lambda) = 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.

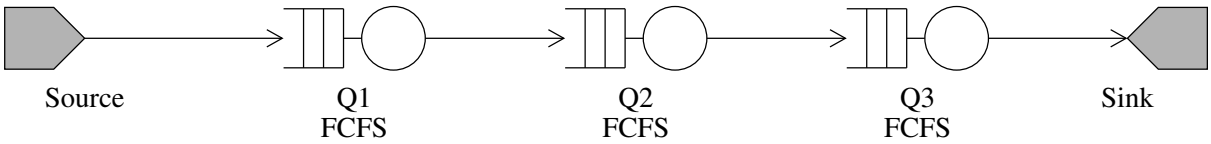

Figure 3.46: Topology of `snc_tandem_multiclass`: an open network over 3 queueing stations.

The example is built and solved as follows.

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`.

```
final double lambda = 0.6;
final double[] rates = {1.5, 1.2, 1.0};
Network model = new Network("SncTandem");
Source source = new Source(model, "Source");
Queue q1 = new Queue(model, "Q1", SchedStrategy.FCFS);
Queue q2 = new Queue(model, "Q2", SchedStrategy.FCFS);
Queue q3 = new Queue(model, "Q3", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Sink");
OpenClass jobclass = new OpenClass(model, "Class1");
source.setArrival(jobclass, new Exp(lambda));
q1.setService(jobclass, new Exp(rates[0]));
q2.setService(jobclass, new Exp(rates[1]));
q3.setService(jobclass, new Exp(rates[2]));
RoutingMatrix P = model.initRoutingMatrix();
P.set(jobclass, jobclass, Network.serialRouting(source, q1, q2, q3, sink));
model.link(P);

NetworkAvgTable tandemTable = new SolverBA(model, "snc.upper").getAvgTable();
tandemTable.print();

// Each hop replaces the arrival envelope by the DEPARTURE envelope of
// the station upstream, which carries the burst the server has added.
// The exact answer does not degrade this way -- by Burke's theorem the
// departure process of an M/M/1 is again Poisson -- so the ratio to the
// exact response time grows hop by hop. The bound stays valid; it is the
// price of assuming nothing about the departure process.

    ... (55 source lines omitted; full implementation: jar/src/main/java/jline/examples/)


SolverBA sharedSolver = new SolverBA(shared, "snc.upper");
sharedSolver.getAvgTable().print();
sharedSolver.getPercTable(1e-3).print();
System.out.printf("exact per-class response time (aggregate M/M/1, lambda=0.6, mu=1): %.4f%n"
    ,
        1.0 / (1.0 - 0.6));
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

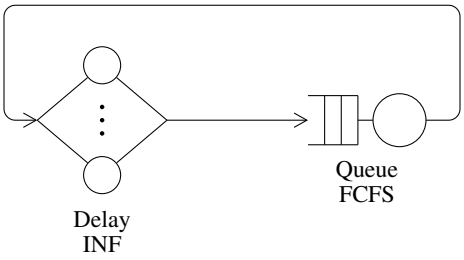

Figure 3.47: Topology of example_mapqn2renv: a closed network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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
Network baseModel = new Network("BaseModel");
Delay delay = new Delay(baseModel, "ThinkTime");
Queue queue = new Queue(baseModel, "Fast/Slow Server", SchedStrategy.FCFS);

// Closed class with 5 jobs
int N = 5;
ClosedClass jobclass = new ClosedClass(baseModel, "Jobs", N, delay, 0);
delay.setService(jobclass, new Exp(1.0));  // Think time = 1.0
queue.setService(jobclass, new Exp(2.0));  // Placeholder service rate

// Connect nodes in a cycle
baseModel.link(Network.serialRouting(delay, queue));

// Block 2: Create the random environment
int E = 2;
Environment env = new Environment("ServerModes", E);
```

```
// Stage 0: Fast mode (service rate = 4.0)
Network fastModel = baseModel.copy();
Queue fastQueue = (Queue) fastModel.getNodeByName("Fast/Slow Server");
fastQueue.setService(fastModel.getClasses().get(0), new Exp(4.0));
env.addStage(0, "Fast", "operational", fastModel);

// Stage 1: Slow mode (service rate = 1.0)
Network slowModel = baseModel.copy();

    ... (36 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

for (int e = 0; e < E; e++) {
    System.out.println("\nStage " + e + ":");
    new MVA(env.getModel(e)).getAvgTable().print();
}

return env;
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.11.3 `renv_container_terminal`

A container terminal case study: MAP arrivals, multiserver and load-dependent stations, several environment stages.

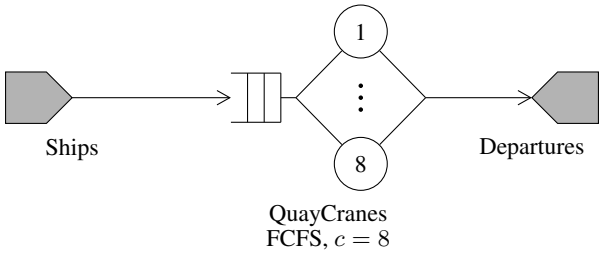

Figure 3.48: Topology of `renv_container_terminal`: an open network over 1 queueing station.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.11.4 `renv_fourstages_repairmen`

The four-stage version.
The example is built and solved as follows.

```
Environment envModel = RandomEnvironmentModel.renv_fourstages_repairmen();
int E = envModel.getEnsemble().size();
```

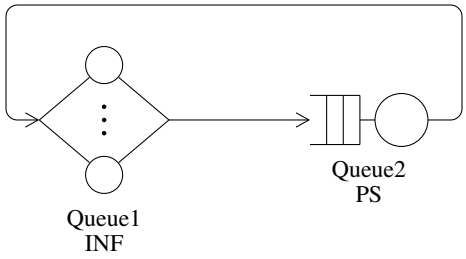

Figure 3.49: Topology of `renv_fourstages_repairmen`: a closed network over 2 queueing stations.

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`.

```
// THIS FIXED POINT IS TOLERANCE-DEPENDENT: run to convergence it lands
// on Queue1 Tput 0.97136 where the reference's iter_tol = 0.05 stops it
// at 0.9716, and the golden holds the latter. So the controls are stated
// rather than left at the engine default.
SolverOptions options = new SolverOptions(SolverType.ENV);
options.timespan = new double[]{0, Double.POSITIVE_INFINITY};
options.iter_max = 100;
options.iter_tol = 0.05;
options.method = "default";
options.verbose = VerboseLevel.STD;

SolverOptions fluidOptions = new SolverOptions(SolverType.FLUID);
fluidOptions.timespan = new double[]{0, Double.POSITIVE_INFINITY};
fluidOptions.verbose = VerboseLevel.SILENT;

// Create solvers for each stage
NetworkSolver[] solvers = new NetworkSolver[E];
for (int e = 0; e < E; e++) {
    solvers[e] = new FLD(envModel.getModel(e));
    solvers[e].options = fluidOptions;
}

// Create environment solver

    ... (3 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    solver.getAvgTable().print();
} catch (Exception e) {
    e.printStackTrace();
}

pauseForUser();
```

Running this edition prints

```
jline.lang.Environment@4f2410ac
```

3.11.5 `renv_genqn`

A randomly generated network placed in an environment.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

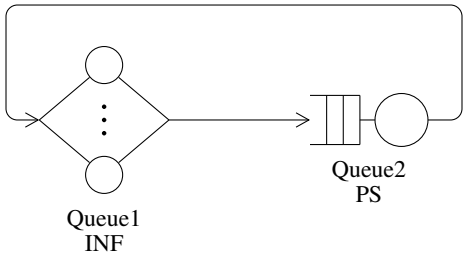

Figure 3.50: Topology of `renv_genqn`: a closed network over 2 queueing stations.

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`.

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.11.7 `renv_map_fallback`

An MMPP(2) stage transition process, and the fallback taken when it cannot be represented exactly.

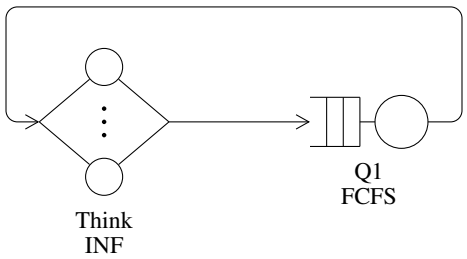

Figure 3.51: Topology of `renv_map_fallback`: a closed network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.11.10 `renv_threestages_repairmen`

The three-stage version.

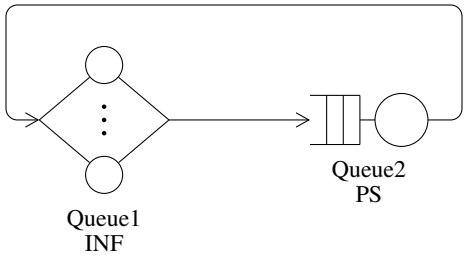

Figure 3.52: Topology of `renv_threestages_repairmen`: a closed network over 2 queueing stations.

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.

```
Environment envModel = RandomEnvironmentModel.renv_threestages_repairmen();
int E = envModel.getEnsemble().size();

// CTMC STAGES, as the reference pins: each stage is solved exactly and
// the horizon is the whole line, so the stage solve is a steady state.
SolverOptions envOptions = new SolverOptions(SolverType.ENV);
envOptions.timespan = new double[]{0, Double.POSITIVE_INFINITY};
envOptions.iter_max = 100;
envOptions.iter_tol = 0.05;
envOptions.method = "default";

// The stage horizon stays at the CTMC default, which is open at both
// ends: SolverENV reads timespan[1] to choose between a steady-state and
// a transient stage solve, and an open horizon is the steady state the
// reference asks each stage for.
SolverOptions ctmcOptions = new SolverOptions(SolverType.CTMC);
ctmcOptions.stiff = false;
ctmcOptions.verbose = VerboseLevel.SILENT;

// Create solvers for each stage
NetworkSolver[] solvers = new NetworkSolver[E];
for (int e = 0; e < E; e++) {
    solvers[e] = new CTMC(envModel.getModel(e));
    solvers[e].options = ctmcOptions;
}


    ... (4 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    solver.getAvgTable().print();
} catch (Exception e) {
    e.printStackTrace();
}

pauseForUser();
```

Running this edition prints

```
ENV analysis [method: default; type: approximate, deterministic; lang: java; env: python] completed in <t>
    s. Iterations: 3.
Station  JobClass     QLen     Util     RespT  ResidT  ArvR    Tput
Queue1   Class1    0.83071  0.83071  0.62313       0     0  1.3331
Queue2   Class1     1.1693  0.75799    0.857       0     0  1.3644

[Running in non-interactive mode, continuing...]
```

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.

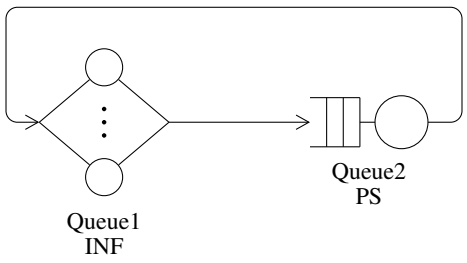

Figure 3.53: Topology of `renv_twostages_repairmen`: a closed network over 2 queueing stations.

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.

```
Environment envModel = RandomEnvironmentModel.renv_twostages_repairmen();
int E = envModel.getEnsemble().size();

// THE ITERATION CONTROLS AND THE STAGE HORIZON ARE PART OF THE GOLDEN.
// An environment couples TRANSIENT stage solves, so a row given a
// different horizon or integrator answers a different question: the
// reference states timespan [0, 1e3] on the fluid solver and leaves the
// integrator at its default, and overriding `stiff` or the ODE max step
// walks a different trajectory.
SolverOptions options = new SolverOptions(SolverType.ENV);
options.timespan = new double[]{0, Double.POSITIVE_INFINITY};
options.iter_max = 100;
options.iter_tol = 0.01;
options.method = "default";
options.verbose = VerboseLevel.STD;

SolverOptions fluidOptions = new SolverOptions(SolverType.FLUID);
fluidOptions.timespan = new double[]{0, 1000};
fluidOptions.verbose = VerboseLevel.SILENT;

// Create solvers for each stage
NetworkSolver[] solvers = new NetworkSolver[E];
for (int e = 0; e < E; e++) {
    solvers[e] = new FLD(envModel.getModel(e));
    solvers[e].options = fluidOptions;
}

    ... (5 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    solver.getAvgTable().print();
} catch (Exception e) {
    e.printStackTrace();
}

pauseForUser();
```

Running this edition prints

```
ENV analysis [method: default; type: approximate, deterministic; lang: java; env: python] completed in <t>
     s. Iterations: 5.
Station  JobClass     QLen     Util    RespT  ResidT  ArvR     Tput
Queue1   Class1    0.55553  0.55553  0.80361       0     0   0.6913
Queue2   Class1    0.44447  0.44447  0.64288       0     0  0.69137

[Running in non-interactive mode, continuing...]
```

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.

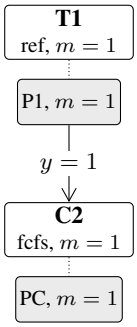

Figure 3.54: Call graph of `lcq_singlehost`: 2 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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.

```
LayeredNetwork model = LayeredCQModel.lcq_singlehost();

// MVA LAYERS, as the reference pins: MVA layers and NC layers are
// different fixed points, and on this model they part company by 4.5%.
LN solver = new LN(model, (subModel) -> new jline.solvers.mva.MVA(subModel, "verbose",
        jline.VerboseLevel.SILENT));

try {
    solver.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
Iter  1.  Analyze time: 0.171s. Update time: 0.013s. Runtime: 0.172s.
Iter  2.  Analyze time: 0.030s. Update time: 0.009s. Runtime: 0.215s.
Iter  3.  Analyze time: 0.029s. Update time: 0.009s. Runtime: 0.254s.
Iter  4.  Analyze time: 0.026s. Update time: 0.010s. Runtime: 0.290s.
Iter  5.  Analyze time: 0.028s. Update time: 0.010s. Runtime: 0.329s.
Iter  6.  Analyze time: 0.027s. Update time: 0.018s. Runtime: 0.368s.
Iter  7.  Analyze time: 0.028s. Update time: 0.009s. Runtime: 0.414s.
Iter  8.  Analyze time: 0.026s. Update time: 0.008s. Runtime: 0.451s.
Iter  9.  Analyze time: 0.028s. Update time: 0.010s. Runtime: 0.489s.
Iter 10.  Analyze time: 0.023s. Update time: 0.008s. Runtime: 0.522s.
Iter 11.  Analyze time: 0.022s. Update time: 0.006s. Runtime: 0.554s.
Iter 12.  Analyze time: 0.017s. Update time: 0.007s. Runtime: 0.578s.
Iter 13.  Analyze time: 0.020s. Update time: 0.006s. Runtime: 0.607s.
Iter 14.  Analyze time: 0.016s. Update time: 0.006s. Runtime: 0.630s.
Iter 15.  Analyze time: 0.017s. Update time: 0.006s. Runtime: 0.653s.
Iter 16.  Analyze time: 0.019s. Update time: 0.006s. Runtime: 0.679s.

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

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

[Running in non-interactive mode, continuing...]
```

3.12.2 `lcq_threehosts`

The three-processor version, where the cache and its clients are on separate hosts.

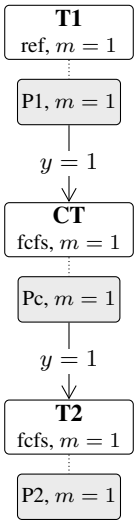

Figure 3.55: Call graph of `lcq_threehosts`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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`.

The example is built and solved as follows.

```
LayeredNetwork model = LayeredCQModel.lcq_threehosts();

// NC layers FIRST, then MVA layers: the reference runs both, and the
// golden holds the first table (goldens/corpus.json multiModelExamples).
LN ncLayers = new LN(model, (subModel) -> new jline.solvers.nc.NC(subModel, "verbose",
        jline.VerboseLevel.SILENT));
LN mvaLayers = new LN(model, (subModel) -> new jline.solvers.mva.MVA(subModel, "verbose",
        jline.VerboseLevel.SILENT));

try {
    ncLayers.getAvgTable().print();
    mvaLayers.getAvgTable().print();
} catch (Exception e) {
}

pauseForUser();
```

Running this edition prints

```
Iter  1.  Analyze time: 0.386s. Update time: 0.016s. Runtime: 0.388s.
Iter  2.  Analyze time: 0.202s. Update time: 0.018s. Runtime: 0.606s.
Iter  3.  Analyze time: 0.173s. Update time: 0.013s. Runtime: 0.798s.
Iter  4.  Analyze time: 0.154s. Update time: 0.013s. Runtime: 0.967s.
Iter  5.  Analyze time: 0.199s. Update time: 0.022s. Runtime: 1.179s.
Iter  6.  Analyze time: 0.222s. Update time: 0.016s. Runtime: 1.426s.
Iter  7.  Analyze time: 0.173s. Update time: 0.012s. Runtime: 1.616s.
Iter  8.  Analyze time: 0.171s. Update time: 0.017s. Runtime: 1.801s.
Iter  9.  Analyze time: 0.180s. Update time: 0.015s. Runtime: 1.999s.
Iter 10.  Analyze time: 0.221s. Update time: 0.011s. Runtime: 2.236s.
Iter 11.  Analyze time: 0.206s. Update time: 0.010s. Runtime: 2.454s.
Iter 12.  Analyze time: 0.186s. Update time: 0.009s. Runtime: 2.651s.
Iter 13.  Analyze time: 0.140s. Update time: 0.009s. Runtime: 2.801s.
Iter 14.  Analyze time: 0.111s. Update time: 0.009s. Runtime: 2.922s.
Iter 15.  Analyze time: 0.103s. Update time: 0.007s. Runtime: 3.035s.
Iter 16.  Analyze time: 0.093s. Update time: 0.008s. Runtime: 3.136s.

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

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

[Running in non-interactive mode, continuing...]
```

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.

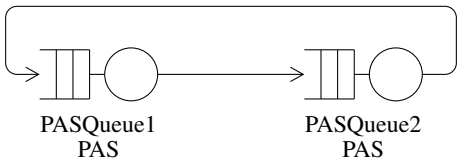

Figure 3.56: Topology of `pas_closed_tandem_fig6`: a closed network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.13.2 pas_compatibility_5class

Five classes and the compatibility matrix that says which may swap with which.

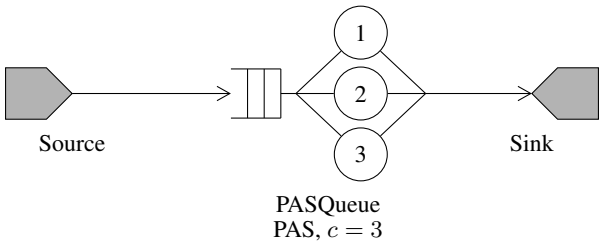

Figure 3.57: Topology of pas_compatibility_5class: an open network over 1 queueing station.

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.

The example is built and solved as follows.

```
Network model = compatibilityModel();
SolverOptions opt = SolverCTMC.defaultOptions();
opt.method = "exact";   // pin the state-space path
opt.cutoff = Matrix.singleton(3);
return new SolverCTMC(model, opt).getAvgTable();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 3 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
jline.solvers.NetworkAvgTable@74f0ea28
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.13.5 pas_mmk

The pass-and-swap discipline on an M/M/k station.

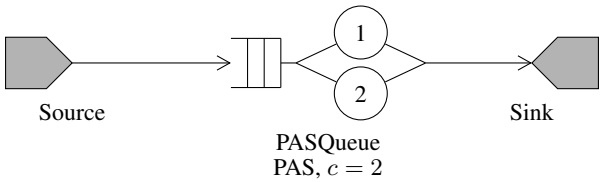

Figure 3.58: Topology of `pas_mmk`: an open network over 1 queueing station.

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.

```
Network model = mmkModel();
SolverOptions opt = SolverCTMC.defaultOptions();
opt.method = "exact";   // pin the state-space path
opt.cutoff = Matrix.singleton(4);
return new SolverCTMC(model, opt).getAvgTable();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 4 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
jline.solvers.NetworkAvgTable@61d47554
```

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.13.8 pas_selfloop

A self-looping class under pass-and-swap.

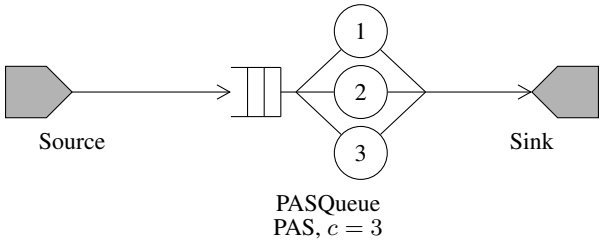

Figure 3.59: Topology of `pas_selfloop`: an open network over 1 queueing station.

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.

```
Network model = selfloopModel();
SolverOptions opt = SolverCTMC.defaultOptions();
opt.method = "exact";   // pin the state-space path
opt.cutoff = Matrix.singleton(3);
return new SolverCTMC(model, opt).getAvgTable();
```

Running this edition prints

```
WARNING: [runAnalyzerBody] CTMC solver using state space cutoff = 3 for open/mixed model. State space
    truncation may cause inaccurate results. Consider varying cutoff to assess sensitivity.
CTMC analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
jline.solvers.NetworkAvgTable@69b794e2
```

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.

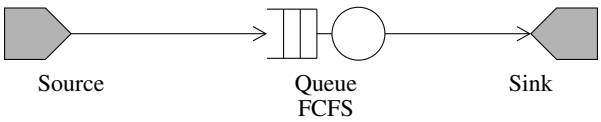

Figure 3.60: Topology of `rewardModel_aggregation`: an open network over 1 queueing station.

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.

```
Node[] n = new Node[3];
Network model = rewardModel("RewardAggregationExample", SchedStrategy.FCFS, 1, 4, n);
final Queue queue = (Queue) n[1];
OpenClass c1 = new OpenClass(model, "HighPriority", 0);
OpenClass c2 = new OpenClass(model, "LowPriority", 0);
((Source) n[0]).setArrival(c1, new Exp(1.0));
((Source) n[0]).setArrival(c2, new Exp(0.8));
queue.setService(c1, new Exp(3.0));
queue.setService(c2, new Exp(3.0));
model.link(Network.serialRouting(model.getClasses(), n[0], n[1], n[2]));

final int i1 = model.getJobClassIndex(c1);
final int i2 = model.getJobClassIndex(c2);
model.setReward("TotalJobs", Reward.queueLength(queue));
```

```
model.setReward("MaxClass", new RewardFunction() {
    private static final long serialVersionUID = 1L;

    public double compute(Matrix s, NetworkStruct sn) {
        return Math.max(at(s, sn, queue, i1), at(s, sn, queue, i2));
    }
});
model.setReward("ClassCount", new RewardFunction() {
    private static final long serialVersionUID = 1L;

    public double compute(Matrix s, NetworkStruct sn) {
        return (at(s, sn, queue, i1) > 0 ? 1.0 : 0.0)

    ... (8 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    public double compute(Matrix s, NetworkStruct sn) {
        return 2.0 * at(s, sn, queue, i1) + at(s, sn, queue, i2);
    }
});

reportSteadyState(new SolverCTMC(model));
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.14.2 `rewardModel_mm1k`

State rewards on an M/M/1/K queue: cost per waiting job, revenue per completion, penalty per loss.

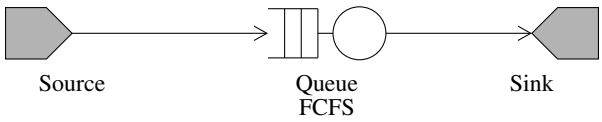

Figure 3.61: Topology of `rewardModel_mm1k`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Node[] n = new Node[3];
Network model = rewardModel("RewardExample", SchedStrategy.FCFS, 1, 3, n);
final Queue queue = (Queue) n[1];
final OpenClass oclass = new OpenClass(model, "Class1", 0);
((Source) n[0]).setArrival(oclass, new Exp(2.0));
queue.setService(oclass, new Exp(3.0));
model.link(Network.serialRouting(n[0], n[1], n[2]));

final int c = model.getJobClassIndex(oclass);
model.setReward("QueueLength", Reward.queueLength(queue, oclass));
model.setReward("Utilization", Reward.utilization(queue, oclass));
model.setReward("BlockingProb", Reward.blocking(queue));
model.setReward("QueueCost", new RewardFunction() {
    private static final long serialVersionUID = 1L;
```

```
    public double compute(Matrix state, NetworkStruct sn) {
        double jobs = at(state, sn, queue, c);
        return jobs * jobs;
    }
});

SolverCTMC solver = new SolverCTMC(model);
reportSteadyState(solver);

Print.section("CTMC transient, timespan [0, 5]");
SolverOptions topt = SolverCTMC.defaultOptions();

    ... (25 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

}
Map<String, Double> R = solver.getAvgReward();
compare("QueueLength", R.get("QueueLength"), L);
compare("Utilization", R.get("Utilization"), 1.0 - pi[0]);
compare("BlockingProb", R.get("BlockingProb"), pi[kbuf]);
compare("QueueCost", R.get("QueueCost"), cost);
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.14.3 `rewardModel_multiclass`

Per-class rewards on a multiclass PS station.

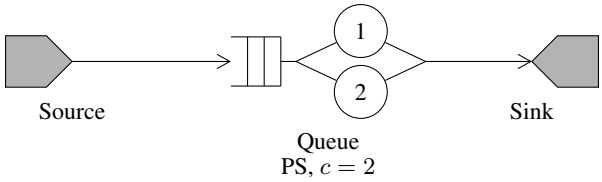

Figure 3.62: Topology of `rewardModel_multiclass`: an open network over 1 queueing station.

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.

```
Node[] n = new Node[3];
Network model = rewardModel("MultiClassRewardExample", SchedStrategy.PS, 2, 6, n);
final Queue queue = (Queue) n[1];
OpenClass ci = new OpenClass(model, "Interactive", 0);
OpenClass cb = new OpenClass(model, "Batch", 0);
((Source) n[0]).setArrival(ci, new Exp(2.0));
((Source) n[0]).setArrival(cb, new Exp(1.5));
queue.setService(ci, new Exp(0.5));
queue.setService(cb, new Exp(1.0));
model.link(Network.serialRouting(model.getClasses(), n[0], n[1], n[2]));

final int i1 = model.getJobClassIndex(ci);
final int i2 = model.getJobClassIndex(cb);
model.setReward("Interactive_QLen", Reward.queueLength(queue, ci));
model.setReward("Interactive_Util", Reward.utilization(queue, ci));
```

```
model.setReward("Batch_QLen", Reward.queueLength(queue, cb));
model.setReward("Batch_Util", Reward.utilization(queue, cb));
model.setReward("Total_QLen", Reward.queueLength(queue));
model.setReward("Total_Util", Reward.utilization(queue));
model.setReward("Interactive_Ratio", new RewardFunction() {
    private static final long serialVersionUID = 1L;

    public double compute(Matrix s, NetworkStruct sn) {
        return at(s, sn, queue, i1) / Math.max(at(s, sn, queue, i2), 0.001);
    }
});

    ... (30 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

if (R.get("Total_QLen") > 0) {
    Print.kv("Interactive share (%)", 100.0 * R.get("Interactive_QLen") / R.get("Total_QLen")
    );
    Print.kv("Batch share (%)", 100.0 * R.get("Batch_QLen") / R.get("Total_QLen"));
}
Print.kv("Interactive resp. time", R.get("Interactive_QLen") / lambdaInt);
Print.kv("Batch resp. time", R.get("Batch_QLen") / lambdaBatch);
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

3.14.4 `rewardModel_templates`

The reward templates that cover the common cases without writing a reward function.

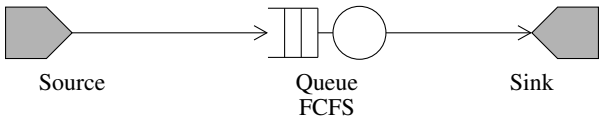

Figure 3.63: Topology of `rewardModel_templates`: an open network over 1 queueing station.

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.

```
Node[] n = new Node[3];
Network model = rewardModel("RewardTemplatesExample", SchedStrategy.FCFS, 1, 10, n);
Queue queue = (Queue) n[1];
OpenClass oclass = new OpenClass(model, "Class1", 0);
((Source) n[0]).setArrival(oclass, new Exp(1.5));
queue.setService(oclass, new Exp(2.0));
model.link(Network.serialRouting(n[0], n[1], n[2]));

model.setReward("QueueLength", Reward.queueLength(queue));
model.setReward("QueueLength_Class1", Reward.queueLength(queue, oclass));
model.setReward("Utilization", Reward.utilization(queue));
model.setReward("Utilization_Class1", Reward.utilization(queue, oclass));
model.setReward("BlockingProb", Reward.blocking(queue));

SolverCTMC solver = new SolverCTMC(model);
reportSteadyState(solver);
```

```
Print.section("M/M/1/K analytical");
double rho = 1.5 / 2.0;
int kbuf = 10;
double[] pi = mm1kPi(rho, kbuf);
double L = 0;
for (int j = 0; j <= kbuf; j++) {
    L += j * pi[j];
}
Map<String, Double> R = solver.getAvgReward();
compare("QueueLength", R.get("QueueLength"), L);
compare("Utilization", R.get("Utilization"), 1.0 - pi[0]);
compare("BlockingProb", R.get("BlockingProb"), pi[kbuf]);
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its 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.

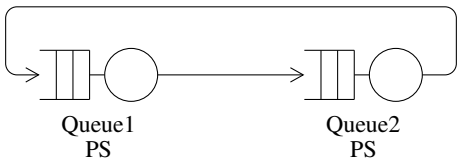

Figure 3.64: Topology of `ag_closed_network`: a closed network over 2 queueing stations.

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.

```
System.out.println("=== Closed Network (2 PS Queues, 10 jobs) ===\n");

Network model = AgentModel.closedNetwork();

// AG with INAP method
System.out.println("AG (method=inap):");
NetworkSolver solverInap = new AG(model, "inap");
solverInap.getAvgTable().print();

// AG with exact method
System.out.println("\nAG (method=exact):");
NetworkSolver solverExact = new AG(model, "exact");
solverExact.getAvgTable().print();

// MVA for comparison
System.out.println("\nMVA:");
```

```
NetworkSolver solverMVA = new MVA(model);
solverMVA.getAvgTable().print();

// CTMC for exact results
System.out.println("\nCTMC (exact):");
NetworkSolver solverCTMC = new CTMC(model);
solverCTMC.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
=== Closed Network (2 PS Queues, 10 jobs) ===

AG (method=inap):
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 91.
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

AG (method=exact):
WARNING: [solver_ag] AutoCAT (exact method) is not available. Falling back to INAP.
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 91.
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:

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


CTMC (exact):
CTMC analysis [method: default; type: exact, deterministic; lang: java; env: python] completed in <t>s.
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

[Running in non-interactive mode, continuing...]
```

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.

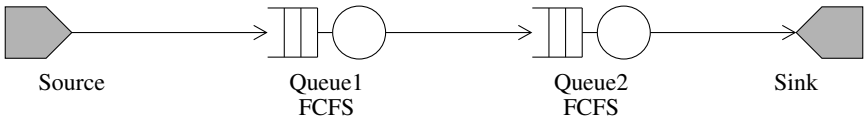

Figure 3.65: Topology of `ag_gnetwork`: an open network over 2 queueing stations.

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.

```
System.out.println("=== G-Network (Gelenbe Network) with Negative Customers ===");
```

```
System.out.println("Positive arrival rate: 1.0");
System.out.println("Negative signal rate: 0.3");
System.out.println("Service rates: mu1=2.0, mu2=3.0\n");

Network model = AgentModel.gNetwork();

// AG with INAP method
System.out.println("AG (method=inap):");
NetworkSolver solverInap = new AG(model, "inap");
solverInap.getAvgTable().print();

// Theoretical insight
System.out.println("\nNote: In G-networks, negative customers reduce the effective");
System.out.println("      load at target queues by removing jobs upon arrival.");
System.out.println("      The utilization at Queue2 is lower than it would be");
System.out.println("      without negative signals due to job removals.");

pauseForUser();
```

Running this edition prints

```
=== G-Network (Gelenbe Network) with Negative Customers ===
Positive arrival rate: 1.0
Negative signal rate: 0.3
Service rates: mu1=2.0, mu2=3.0

AG (method=inap):
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 2.
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

Note: In G-networks, negative customers reduce the effective
      load at target queues by removing jobs upon arrival.
      The utilization at Queue2 is lower than it would be
      without negative signals due to job removals.

[Running in non-interactive mode, continuing...]
```

3.15.3 ag_jackson_network

A Jackson network built agent by agent.

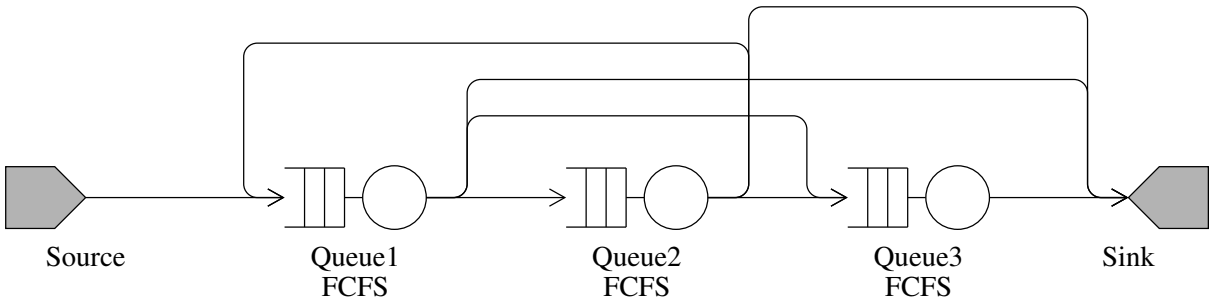

Figure 3.66: Topology of ag_jackson_network: an open network over 3 queueing stations.

The example is built and solved as follows.

```
System.out.println("=== Jackson Network (3 Queues) ===\n");

Network model = AgentModel.jacksonNetwork();

// AG with INAP method
System.out.println("AG (method=inap):");
```

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`.

```
NetworkSolver solverInap = new AG(model, "inap");
solverInap.getAvgTable().print();

// AG with exact method
System.out.println("\nAG (method=exact):");
NetworkSolver solverExact = new AG(model, "exact");
solverExact.getAvgTable().print();

// MVA for comparison
System.out.println("\nMVA:");
NetworkSolver solverMVA = new MVA(model);
solverMVA.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
=== Jackson Network (3 Queues) ===

AG (method=inap):
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 8.
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

AG (method=exact):
WARNING: [solver_ag] AutoCAT (exact method) is not available. Falling back to INAP.
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 8.
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

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

MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 54.
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

[Running in non-interactive mode, continuing...]
```

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.
The example is built and solved as follows.

```
System.out.println("=== Multiclass Closed Network ===");
System.out.println("Class 1: 5 jobs, Class 2: 3 jobs\n");

Network model = AgentModel.multiclassClosed();
```

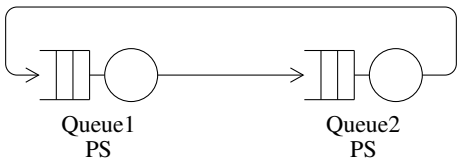

Figure 3.67: Topology of `ag_multiclass_closed`: a closed network over 2 queueing stations.

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`.

```
// AG with INAP method
System.out.println("AG (method=inap):");
NetworkSolver solverInap = new AG(model, "inap");
solverInap.getAvgTable().print();

// AG with exact method
System.out.println("\nAG (method=exact):");
NetworkSolver solverExact = new AG(model, "exact");
solverExact.getAvgTable().print();

// MVA for comparison
System.out.println("\nMVA:");
NetworkSolver solverMVA = new MVA(model);
solverMVA.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
=== Multiclass Closed Network ===
Class 1: 5 jobs, Class 2: 3 jobs

AG (method=inap):
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 49.
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

AG (method=exact):
WARNING: [solver_ag] AutoCAT (exact method) is not available. Falling back to INAP.
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
     Iterations: 49.
Station  JobClass    QLen     Util    RespT  ResidT     ArvR     Tput
Queue1   Class1    1.6243  0.67984  1.1946  1.1946  0.95678   1.3597

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

MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
     s.
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

[Running in non-interactive mode, continuing...]
```

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.

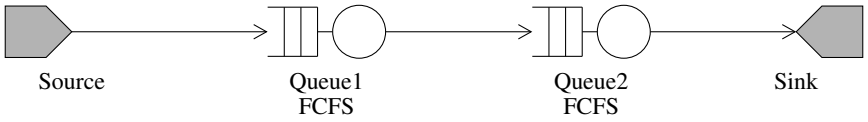

Figure 3.68: Topology of `ag_tandem_open`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
System.out.println("=== Open Tandem Queue (M/M/1 -> M/M/1) ===\n");

Network model = AgentModel.tandemOpen();

// Analytical solution for comparison
double lambda = 0.5;
double mu1 = 1.0;
double mu2 = 1.5;
double U1_exact = lambda / mu1;
double U2_exact = lambda / mu2;
double Q1_exact = U1_exact / (1 - U1_exact);
double Q2_exact = U2_exact / (1 - U2_exact);

System.out.println("Analytical (M/M/1):");
System.out.printf("  Queue1: U=%.4f, Q=%.4f%n", U1_exact, Q1_exact);
System.out.printf("  Queue2: U=%.4f, Q=%.4f%n%n", U2_exact, Q2_exact);

// AG with INAP method
System.out.println("AG (method=inap):");
NetworkSolver solverInap = new AG(model, "inap");
solverInap.getAvgTable().print();

// AG with exact method
System.out.println("\nAG (method=exact):");
NetworkSolver solverExact = new AG(model, "exact");
solverExact.getAvgTable().print();

// MVA for comparison
System.out.println("\nMVA:");
NetworkSolver solverMVA = new MVA(model);
solverMVA.getAvgTable().print();

pauseForUser();
```

Running this edition prints

```
=== Open Tandem Queue (M/M/1 -> M/M/1) ===

Analytical (M/M/1):
  Queue1: U=0.5000, Q=1.0000
  Queue2: U=0.3333, Q=0.5000
```

```
AG (method=inap):
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.
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

AG (method=exact):
WARNING: [solver_ag] AutoCAT (exact method) is not available. Falling back to INAP.
AG analysis [method: inap; type: approximate, deterministic; lang: java; env: python] completed in <t>s.

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

MVA:
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 49.
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

[Running in non-interactive mode, continuing...]
```

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.

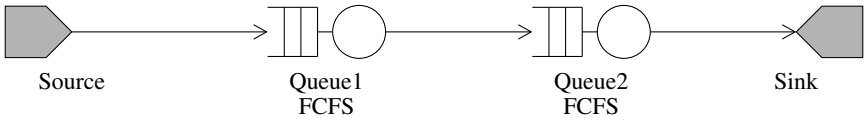

Figure 3.69: Topology of `ag_tandem_phasetype`: an open network over 2 queueing stations.

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.

```
System.out.println("=== Open Tandem with Phase-Type Service ===\n");

Network model = AgentModel.tandemPhaseType();

double lambda = 0.5, meanS = 1.0, scv1 = 0.5;
double rho = lambda * meanS;
double pk1 = rho + rho * rho * (1 + scv1) / (2 * (1 - rho));
System.out.println("Queue1 is an isolated M/Er2/1, so its exact mean queue length is the");
System.out.printf("Pollaczek-Khinchine value %.6f. The M/M/1 reading would be %.6f.%n%n",
        pk1, rho / (1 - rho));

System.out.println("AG (method=inap):");
new AG(model, "inap").getAvgTable().print();

// '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.
System.out.println("\nAG (method=inapinf):");
new AG(model, "inapinf").getAvgTable().print();

pauseForUser();
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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 `jline.examples.java.Gallery`, one static method per model. A gallery entry never prints: it builds the model and returns it, so the caller chooses the solver.

```
public static Network gallery_mm1() {
    Network model = new Network("M/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, new Exp(1));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(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.

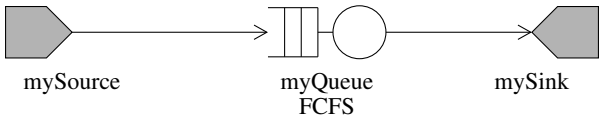

Figure 4.1: Topology of `gallery_aphm1`: an open network over 1 queueing station.

The example is built and solved as follows.

```
Network model = new Network("APH/M/1");
Source source = new Source(model, "mySource");
Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "mySink");
OpenClass oclass = new OpenClass(model, "myClass");
source.setArrival(oclass, APH.fitCentral(1, 0.99, 1.999));
queue.setService(oclass, new Exp(2));
```

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`.

```
model.link(Network.serialRouting(source, queue, sink));
return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.2 `gallery_cache_lru`

A cache node under least-recently-used replacement, read by a closed class.

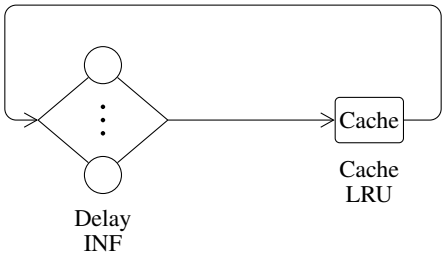

Figure 4.2: Topology of `gallery_cache_lru`: a closed network over 1 queueing station with a cache node.

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.

```
public static Network gallery_cache_lru() {
    Network model = new Network("Cache-LRU");
    int n = 5; // number of items
    int m = 2; // cache capacity
    Delay delay = new Delay(model, "Delay");
    Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.LRU);
    ClosedClass jobClass = new ClosedClass(model, "JobClass", 1, delay, 0);
    ClosedClass hitClass = new ClosedClass(model, "HitClass", 0, delay, 0);
    ClosedClass missClass = new ClosedClass(model, "MissClass", 0, delay, 0);
    delay.setService(jobClass, new Exp(1));
    Matrix pmatrix = new Matrix(1, n);
    for (int i = 0; i < n; i++) {
        pmatrix.set(0, i, 1.0 / n);
    }
    DiscreteSampler pAccess = new DiscreteSampler(pmatrix);
    cacheNode.setRead(jobClass, pAccess);
    cacheNode.setHitClass(jobClass, hitClass);
```

```
    cacheNode.setMissClass(jobClass, missClass);
    RoutingMatrix P = model.initRoutingMatrix();
    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;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.3 `gallery_cache_routing`

A cache whose hit and miss outcomes route the job to different downstream stations.

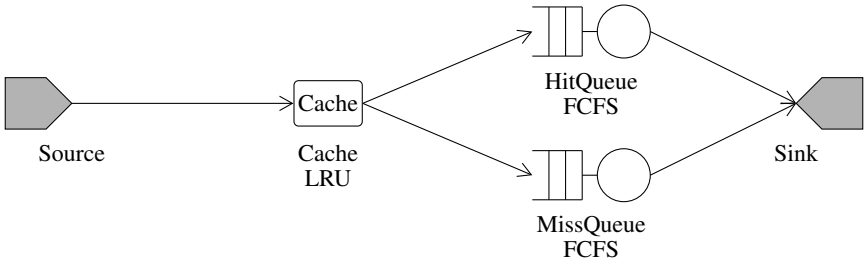

Figure 4.3: Topology of `gallery_cache_routing`: an open network over 2 queueing stations with a cache node.

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.

```
public static Network gallery_cache_routing() {
    Network model = new Network("Cache-Routing");
    int n = 4; // number of items
    int m = 2; // cache capacity
    Source source = new Source(model, "Source");
    Cache cacheNode = new Cache(model, "Cache", n, m, ReplacementStrategy.LRU);
    Queue hitQueue = new Queue(model, "HitQueue", SchedStrategy.FCFS);
    Queue missQueue = new Queue(model, "MissQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Sink");
    OpenClass jobClass = new OpenClass(model, "InitClass", 0);
    OpenClass hitClass = new OpenClass(model, "HitClass", 0);
    OpenClass missClass = new OpenClass(model, "MissClass", 0);
    source.setArrival(jobClass, new Exp(1));
    hitQueue.setService(hitClass, new Exp(2.0));
    missQueue.setService(missClass, new Exp(1.0));
    Matrix pmatrix = new Matrix(1, n);
    for (int i = 0; i < n; i++) {
        pmatrix.set(0, i, 1.0 / n);
    }
```

```
    DiscreteSampler pAccess = new DiscreteSampler(pmatrix);
    cacheNode.setRead(jobClass, pAccess);
    cacheNode.setHitClass(jobClass, hitClass);
    cacheNode.setMissClass(jobClass, missClass);
    RoutingMatrix P = model.initRoutingMatrix();
    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;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.4 gallery_coxm1

The Coxian/M/1 queue: Coxian interarrival times, exponential service, and a single server.

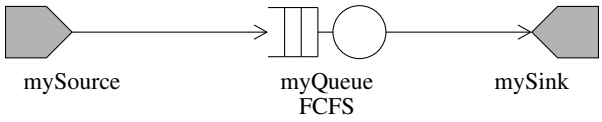

Figure 4.4: Topology of gallery_coxm1: an open network over 1 queueing station.

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.

```
Network model = new Network("Cox/M/1");
Source source = new Source(model, "mySource");
Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "mySink");
OpenClass oclass = new OpenClass(model, "myClass");
source.setArrival(oclass, Coxian.fitCentral(1, 0.99, 1.999));
queue.setService(oclass, new Exp(2));
model.link(Network.serialRouting(source, queue, sink));
return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.5 gallery_cqn

The closed queueing network archetype: a fixed job population circulating between a Delay station and a queue.
The example is built and solved as follows.

```
return gallery_cqn(2, false, 2300);
```

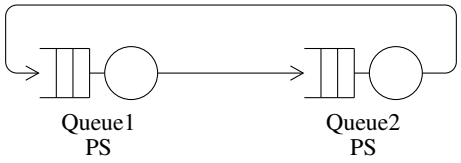

Figure 4.5: Topology of `gallery_cqn`: a closed network over 2 queueing stations.

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`.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.6 `gallery_cqn_multiclass`

The closed archetype with several classes, each with its own population.

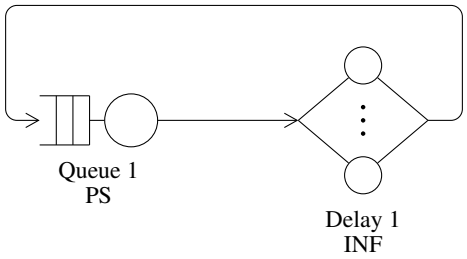

Figure 4.6: Topology of `gallery_cqn_multiclass`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
public static Network gallery_cqn_multiclass() {
    return gallery_cqn_multiclass(2, 2, false, 2300);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.7 `gallery_detm1`

The D/M/1 queue: deterministic interarrival times, exponential service, and a single server.
The example is built and solved as follows.

```
public static Network gallery_detm1() {
    Network model = new Network("D/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, new Det(1));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

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`.

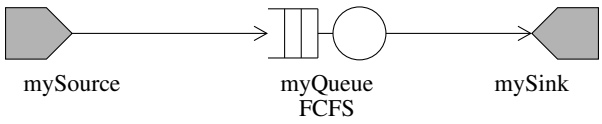

Figure 4.7: Topology of `gallery_detm1`: an open network over 1 queueing station.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.8 `gallery_dm1`

The D/M/1 queue: deterministic interarrival times, exponential service, and a single server.
The example is built and solved as follows.

```
public static Network gallery_dm1() {
    Network model = new Network("D/M/1");
    Source source = new Source(model, "Source");
    Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass = new OpenClass(model, "Class1");
    source.setArrival(oclass, new Det(1));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.9 `gallery_erldk`

The Erl/D/k queue: Erlang interarrival times, deterministic service, and k parallel servers.
The example is built and solved as follows.

```
public static Network gallery_erldk() {
    return gallery_erldk(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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`.

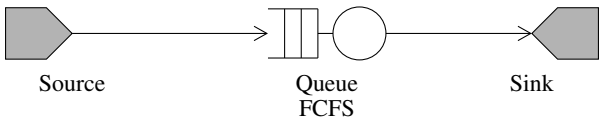

Figure 4.8: Topology of `gallery_dm1`: an open network over 1 queueing station.

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`.

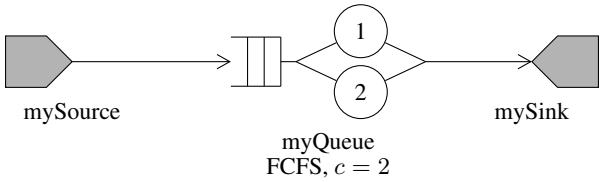

Figure 4.9: Topology of `gallery_erldk`: an open network over 1 queueing station.

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`.

4.2.10 gallery_erlerl1

The Erl/Erl/1 queue: Erlang interarrival times, Erlang service, and a single server.

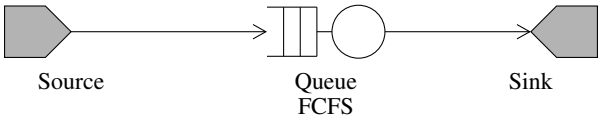

Figure 4.10: Topology of `gallery_erlerl1`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
public static Network gallery_erlerl1() {
    return gallery_erlerl1(5);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.11 gallery_erlerl1_reentrant

An Erl/Erl/1 queue with re-entrant visits.

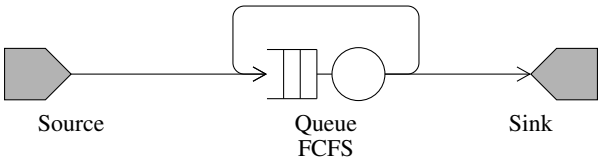

Figure 4.11: Topology of `gallery_erlerl1_reentrant`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_erlerl1_reentrant() {
    return gallery_erlerl1_reentrant(5);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.12 gallery_erlm1

The Erl/M/1 queue: Erlang interarrival times, exponential service, and a single server.
The example is built and solved as follows.

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`.

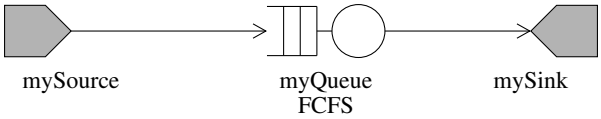

Figure 4.12: Topology of `gallery_erlm1`: an open network over 1 queueing station.

```
public static Network gallery_erlm1() {
    Network model = new Network("Erl/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, Erlang.fitMeanAndOrder(1.0, 5));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.
The example is built and solved as follows.

```
public static Network gallery_erlm1_ps() {
    Network model = new Network("Erl/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.PS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, Erlang.fitMeanAndOrder(1.0, 5));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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`.

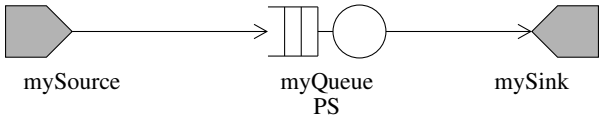

Figure 4.13: Topology of `gallery_erlm1_ps`: an open network over 1 queueing station.

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`.

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.

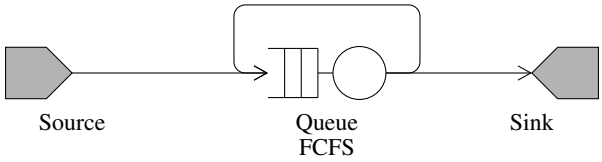

Figure 4.14: Topology of `gallery_erlm1_reentrant`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_erlm1_reentrant() {
    Network model = new Network("Erl/M/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, Erlang.fitMeanAndOrder(1, 5)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, new Exp(2)); // (Queue,Class1)
    node2.setService(jobclass2, new Exp(3)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

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`.

```
no such class: jline.examples.java.Gallery
```

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.

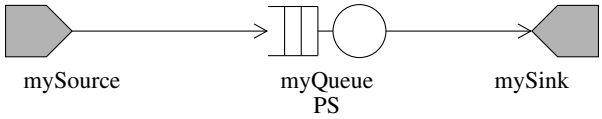

Figure 4.15: Topology of `gallery_erlm1ps`: an open network over 1 queueing station.

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.

```
public static Network gallery_erlm1ps() {
    Network model = new Network("Er/M/1-PS");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.PS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, Erlang.fitMeanAndOrder(1.0, 5));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.
The example is built and solved as follows.

```
public static Network gallery_fcr() {
    return gallery_fcr(3);
```

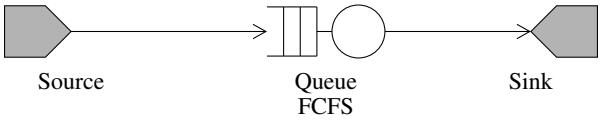

Figure 4.16: Topology of `gallery_fcr`: an open network over 1 queueing station.

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`.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.17 `gallery_fj_closed`

The closed fork-join counterpart, with a fixed population circulating through the block.

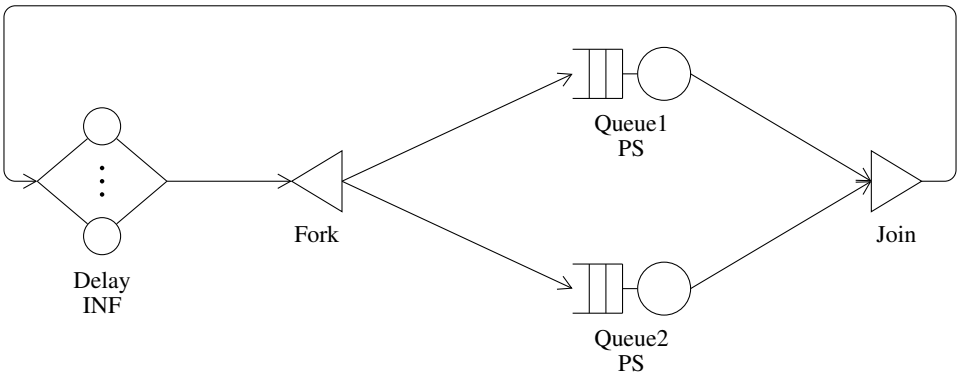

Figure 4.17: Topology of `gallery_fj_closed`: a closed network over 3 queueing stations with a fork-join block.

The example is built and solved as follows.

```
public static Network gallery_fj_closed() {
    Network model = new Network("Fork-Join-Closed");
    Delay delay = new Delay(model, "Delay");
    Queue queue1 = new Queue(model, "Queue1", SchedStrategy.PS);
    Queue queue2 = new Queue(model, "Queue2", SchedStrategy.PS);
    Fork fork = new Fork(model, "Fork");
    Join join = new Join(model, "Join", fork);
    ClosedClass oclass = new ClosedClass(model, "class1", 5, delay);
    delay.setService(oclass, new Exp(1.0));
    queue1.setService(oclass, new Exp(1.0));
    queue2.setService(oclass, new Exp(1.0));
    RoutingMatrix P = model.initRoutingMatrix();
    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);
```

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`.

```
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.18 `gallery_fj_open`

An open fork-join block: a job forks into parallel tasks and its siblings are joined before departure.

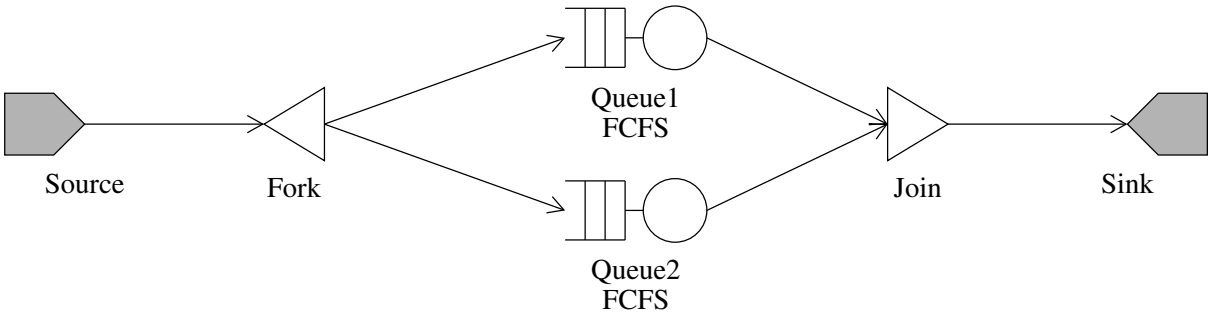

Figure 4.18: Topology of `gallery_fj_open`: an open network over 2 queueing stations with a fork-join block.

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`.

The example is built and solved as follows.

```
public static Network gallery_fj_open() {
    Network model = new Network("Fork-Join-Open");
    Source source = new Source(model, "Source");
    Queue queue1 = new Queue(model, "Queue1", SchedStrategy.FCFS);
    Queue queue2 = new Queue(model, "Queue2", SchedStrategy.FCFS);
    Fork fork = new Fork(model, "Fork");
    Join join = new Join(model, "Join", fork);
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass = new OpenClass(model, "class1");
    source.setArrival(oclass, new Exp(0.05));
    queue1.setService(oclass, new Exp(1.0));
    queue2.setService(oclass, new Exp(2.0));
    RoutingMatrix P = model.initRoutingMatrix();
    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;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

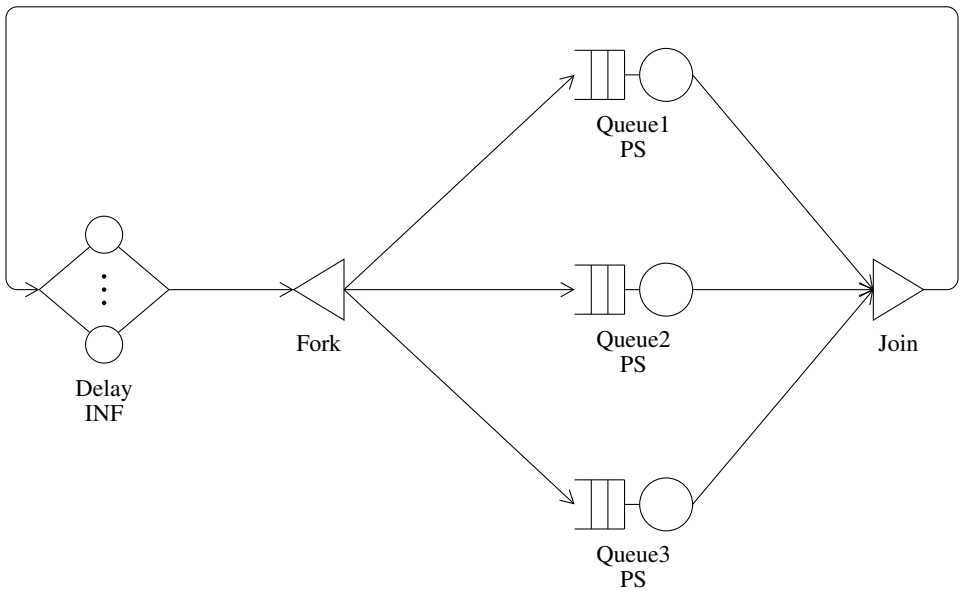

Figure 4.19: Topology of gallery_fj_quorum: a closed network over 4 queueing stations with a fork-join block.

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.

The example is built and solved as follows.

```
Network model = new Network("Fork-Join-Quorum");
Delay delay = new Delay(model, "Delay");
Queue queue1 = new Queue(model, "Queue1", SchedStrategy.PS);
Queue queue2 = new Queue(model, "Queue2", SchedStrategy.PS);
Queue queue3 = new Queue(model, "Queue3", SchedStrategy.PS);
Fork fork = new Fork(model, "Fork");
Join join = new Join(model, "Join", fork);
ClosedClass oclass = new ClosedClass(model, "class1", 5, delay);
delay.setService(oclass, new Exp(1.0));
queue1.setService(oclass, new Exp(2.0));
queue2.setService(oclass, new Exp(2.0));
queue3.setService(oclass, new Exp(2.0));
```

```
join.setStrategy(oclass, JoinStrategy.PARTIAL);
join.setRequired(oclass, 2);
RoutingMatrix P = model.initRoutingMatrix();
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;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.20 gallery_gamm1

The Gamma/M/1 queue: Gamma interarrival times, exponential service, and a single server.

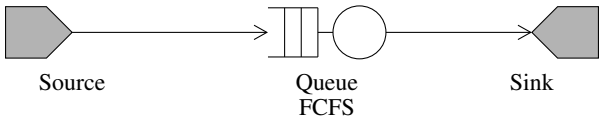

Figure 4.20: Topology of gallery_gamm1: an open network over 1 queueing station.

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.

```
public static Network gallery_gamm1() {
    Network model = new Network("Gam/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, Gamma.fitMeanAndSCV(1.0, 1.0 / 5));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.21 gallery_hyperl1_feedback

A HyperExp/Erl/1 queue with probabilistic feedback.
The example is built and solved as follows.

```
public static Network gallery_hyperl1_feedback() {
```

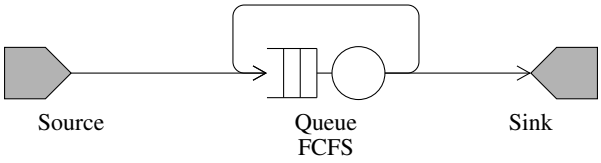

Figure 4.21: Topology of `gallery_hyperl1_feedback`: an open network over 1 queueing station.

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`.

```
    Network model = new Network("Hyper/Erl/1-Feedback");
    Source source = new Source(model, "Source");
    Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass1 = new OpenClass(model, "Class1");
    source.setArrival(oclass1, HyperExp.fitMeanAndSCV(1, 64));
    queue.setService(oclass1, Erlang.fitMeanAndOrder(0.05, 5));
    RoutingMatrix P = model.initRoutingMatrix();
    P.set(oclass1, oclass1, source, queue, 1.0);
    P.set(oclass1, oclass1, queue, queue, 0.9);
    P.set(oclass1, oclass1, queue, sink, 0.1);
    model.link(P);
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

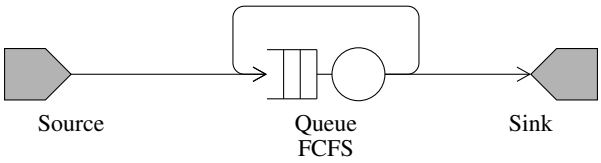

Figure 4.22: Topology of `gallery_hyperl1_reentrant`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_hyperl1_reentrant() {
    Network model = new Network("Hyper/Erl/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
```

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`.

```
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, HyperExp.fitMeanAndSCV(1, 64)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, Erlang.fitMeanAndOrder(0.5, 5)); // (Queue,Class1)
    node2.setService(jobclass2, new Exp(3)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.23 `gallery_hyperlk`

The HyperExp/Erl/k queue: hyperexponential interarrival times, Erlang service, and k parallel servers.

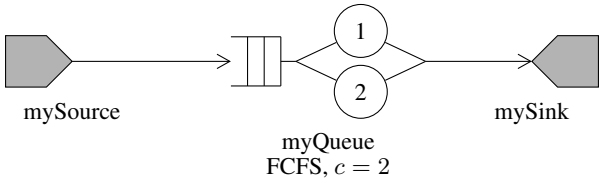

Figure 4.23: Topology of `gallery_hyperlk`: an open network over 1 queueing station.

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.

```
public static Network gallery_hyperlk() {
    return gallery_hyperlk(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

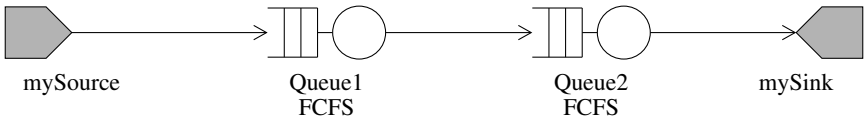

Figure 4.24: Topology of gallery_hyphyp1_linear: an open network over 2 queueing stations.

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.

```
public static Network gallery_hyphyp1_linear() {
    return gallery_hyphyp1_linear(2, 0.9);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

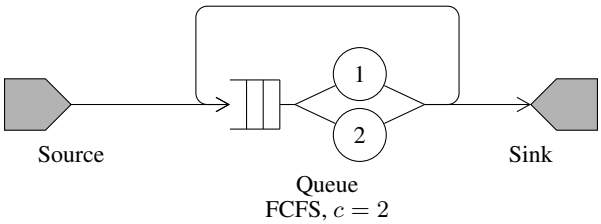

Figure 4.25: Topology of gallery_hyphyp1_reentrant: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_hyphyp1_reentrant() {
    Network model = new Network("Hyper/Hyper/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
```

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`.

```
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    node2.setNumberOfServers(2);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, HyperExp.fitMeanAndSCV(1, 64)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, HyperExp.fitMeanAndSCV(0.5, 4)); // (Queue,Class1)
    node2.setService(jobclass2, new Exp(3)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.26 `gallery_hyphyp1_tandem`

Two HyperExp/HyperExp/1 queues in series.

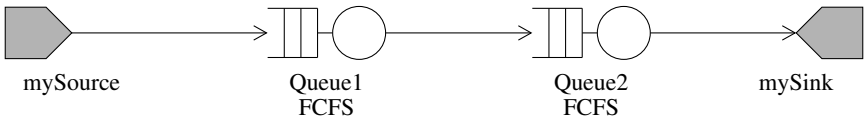

Figure 4.26: Topology of `gallery_hyphyp1_tandem`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
public static Network gallery_hyphyp1_tandem() {
    return gallery_hyphyp1_linear(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.27 gallery_hypm1

The HyperExp/M/1 queue: hyperexponential interarrival times, exponential service, and a single server.

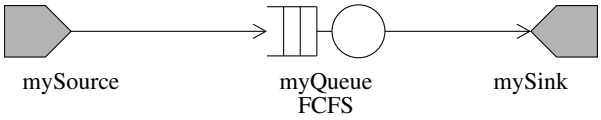

Figure 4.27: Topology of `gallery_hypm1`: an open network over 1 queueing station.

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.

```
public static Network gallery_hypm1() {
    Network model = new Network("H2/M/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, HyperExp.fitMeanAndSCV(1, 64));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

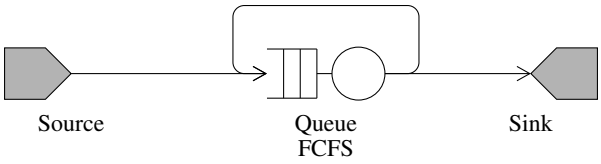

Figure 4.28: Topology of `gallery_hypm1_reentrant`: an open network over 1 queueing station.

The example is built and solved as follows.

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`.

```
public static Network gallery_hypm1_reentrant() {
    Network model = new Network("Hyper/Erl/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, HyperExp.fitMeanAndSCV(1, 4)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, new Exp(2)); // (Queue,Class1)
    node2.setService(jobclass2, new Exp(3)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.29 `gallery_lqn_basic`

The smallest layered queueing network: one reference task calling one server task.

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`.

The example is built and solved as follows.

```
public static LayeredNetwork gallery_lqn_basic() {
    LayeredNetwork model = new LayeredNetwork("LQN-Basic");
    Processor P1 = new Processor(model, "P1", 2, SchedStrategy.PS);
```

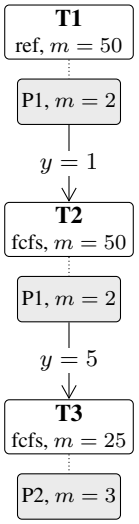

Figure 4.29: Call graph of `gallery_lqn_basic`: 3 tasks over 2 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

```
    Processor P2 = new Processor(model, "P2", 3, SchedStrategy.PS);
    Task T1 = new Task(model, "T1", 50, SchedStrategy.REF).on(P1).setThinkTime(new Exp(1.0 /
    2));
    Task T2 = new Task(model, "T2", 50, SchedStrategy.FCFS).on(P1).setThinkTime(new Exp(1.0 /
     3));
    Task T3 = new Task(model, "T3", 25, SchedStrategy.FCFS).on(P2).setThinkTime(new Exp(1.0 /
     4));
    Entry E1 = new Entry(model, "E1").on(T1);
    Entry E2 = new Entry(model, "E2").on(T2);
    Entry E3 = new Entry(model, "E3").on(T3);
    Activity A1 = new Activity(model, "AS1", new Exp(10)).on(T1).boundTo(E1).synchCall(E2, 1)
    ;
    Activity A2 = new Activity(model, "AS2", new Exp(20)).on(T2).boundTo(E2).synchCall(E3, 5)
    .repliesTo(E2);
    Activity A3 = new Activity(model, "AS3", new Exp(50)).on(T3).boundTo(E3).repliesTo(E3);
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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`.

The example is built and solved as follows.

```
public static LayeredNetwork gallery_lqn_random() {
    return gallery_lqn_random(23000L);
```

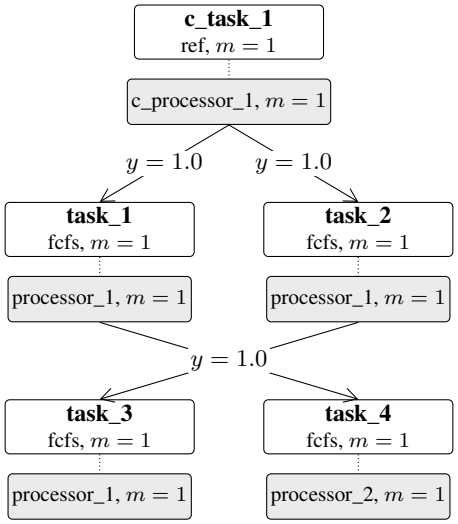

Figure 4.30: Call graph of `gallery_lqn_random`: 5 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.31 `gallery_lqn_workflows`

Workflow patterns (sequence, AND-fork and join, loop) as an activity graph in a layered model.

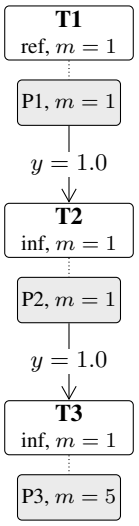

Figure 4.31: Call graph of `gallery_lqn_workflows`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

The example is built and solved as follows.

```
public static LayeredNetwork gallery_lqn_workflows() {
    LayeredNetwork model = new LayeredNetwork("LQN-Workflows");
    Processor P1 = new Processor(model, "P1", Integer.MAX_VALUE, SchedStrategy.INF);
```

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`.

```
    Task T1 = new Task(model, "T1", 1, SchedStrategy.REF).on(P1);
    T1.setThinkTime(new Immediate());
    Entry E1 = new Entry(model, "Entry").on(T1);

    Processor P2 = new Processor(model, "P2", Integer.MAX_VALUE, SchedStrategy.INF);
    Task T2 = new Task(model, "T2", Integer.MAX_VALUE, SchedStrategy.INF).on(P2).setThinkTime
    (new Immediate());
    Entry E2 = new Entry(model, "E2").on(T2);

    Processor P3 = new Processor(model, "P3", 5, SchedStrategy.PS);
    Task T3 = new Task(model, "T3", Integer.MAX_VALUE, SchedStrategy.INF).on(P3);
    T3.setThinkTime(Exp.fitMean(10));
    Entry E3 = new Entry(model, "E3").on(T3);

    Activity A1 = new Activity(model, "A1", Exp.fitMean(1)).on(T1).boundTo(E1);
    Activity A2 = new Activity(model, "A2", Exp.fitMean(2)).on(T1);
    Activity A3 = new Activity(model, "A3", Exp.fitMean(3)).on(T1).synchCall(E2, 1.0);

    Activity B1 = new Activity(model, "B1", Exp.fitMean(0.1)).on(T2).boundTo(E2);
    Activity B2 = new Activity(model, "B2", Exp.fitMean(0.2)).on(T2);
    Activity B3 = new Activity(model, "B3", Exp.fitMean(0.3)).on(T2);
    Activity B4 = new Activity(model, "B4", Exp.fitMean(0.4)).on(T2);
    Activity B5 = new Activity(model, "B5", Exp.fitMean(0.5)).on(T2);
    Activity B6 = new Activity(model, "B6", Exp.fitMean(0.6)).on(T2).synchCall(E3, 1.0).
    repliesTo(E2);

    ... (31 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    List<Activity> orJoinSources = new ArrayList<Activity>();
    orJoinSources.add(C2);
    orJoinSources.add(C3);
    orJoinSources.add(C4);
    T3.addPrecedence(ActivityPrecedence.OrJoin(orJoinSources, C5));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

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`.

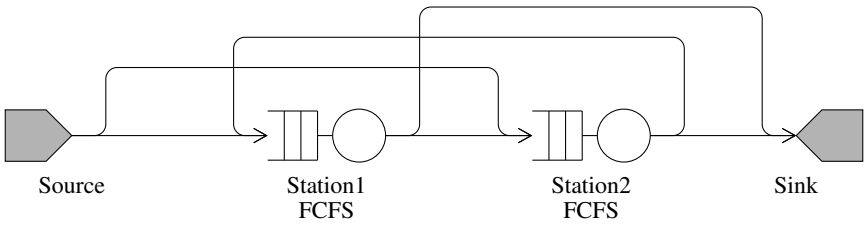

Figure 4.32: Topology of `gallery_lukumar_reentrant`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
public static Network gallery_lukumar_reentrant() {
    return gallery_lukumar_reentrant("FCFS");
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.33 `gallery_mapm1`

The MAP/M/1 queue: a Markovian arrival process, exponential service, and a single server.

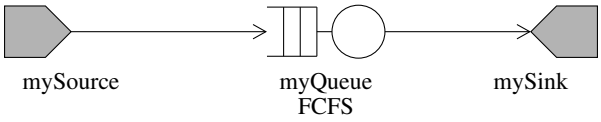

Figure 4.33: Topology of `gallery_mapm1`: an open network over 1 queueing station.

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.

```
public static Network gallery_mapm1() {
    return gallery_mapm1(MAP.rand(2));
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.34 `gallery_mapmk`

The MAP/M/k queue: a Markovian arrival process, exponential service, and k parallel servers.
The example is built and solved as follows.

```
public static Network gallery_mapmk() {
    return gallery_mapmk(MAP.rand(2), 2);
```

Running this edition prints

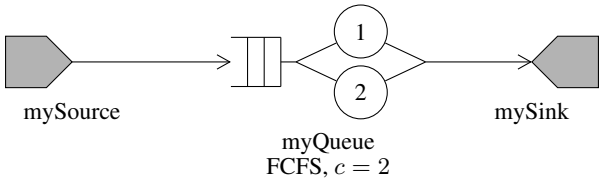

Figure 4.34: Topology of `gallery_mapmk`: an open network over 1 queueing station.

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`.

```
no such class: jline.examples.java.Gallery
```

4.2.35 gallery_mdk

The M/D/k queue: Poisson arrivals, deterministic service, and k parallel servers.

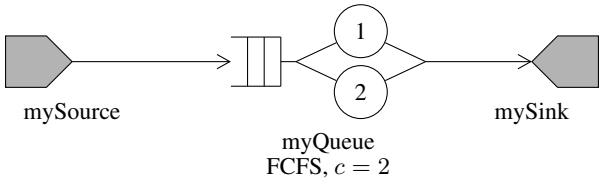

Figure 4.35: Topology of `gallery_mdk`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_mdk() {
    return gallery_mdk(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.36 gallery_merl1

The M/Erl/1 queue: Poisson arrivals, Erlang service, and a single server.
The example is built and solved as follows.

```
Network model = new Network("M/E/1");

// Block 1: nodes
Source source = new Source(model, "mySource");
Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "mySink");

// Block 2: classes
OpenClass oclass = new OpenClass(model, "myClass");
source.setArrival(oclass, new Exp(1.0));
queue.setService(oclass, Erlang.fitMeanAndOrder(0.5, 2));
```

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`.

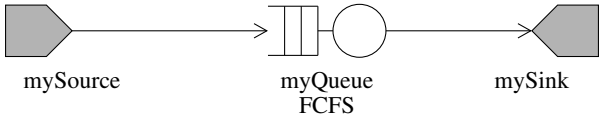

Figure 4.36: Topology of `gallery_merl1`: an open network over 1 queueing station.

```
// Block 3: topology
model.link(Network.serialRouting(source, queue, sink));

return new Object[]{model, source, queue, sink, oclass};
```

Running this edition prints

```
[Ljava.lang.Object;@46ee7fe8
```

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.

```
public static Network gallery_merl1_linear() {
    return gallery_merl1_linear(2, 0.9);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.
The example is built and solved as follows.

```
public static Network gallery_merl1_reentrant() {
    Network model = new Network("M/Erl/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
```

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`.

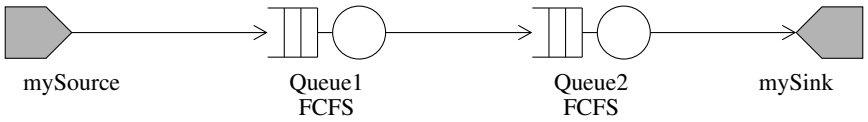

Figure 4.37: Topology of `gallery_merl1_linear`: an open network over 2 queueing stations.

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`.

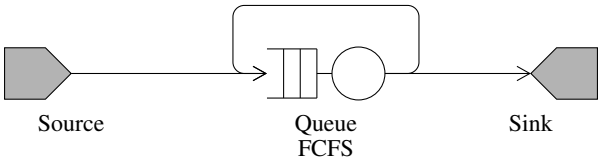

Figure 4.38: Topology of `gallery_merl1_reentrant`: an open network over 1 queueing station.

Nodes	3: Source, Queue, Sink
Classes	2: Class1 (open), Class2 (open)
Scheduling	Queue: FCFS
Arrivals	Source – Class1: Exp(1); Class2: Disabled
Service	Queue – Class1: Erlang(10,5); Class2: Exp(3)
Features	class switching on 1 link

Table 4.39: Features of `gallery_merl1_reentrant`.

```
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, Exp.fitMean(1.00)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, Erlang.fitMeanAndOrder(0.5, 5)); // (Queue,Class1)
    node2.setService(jobclass2, Exp.fitMean(0.333333)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.39 gallery_merl1_tandem

Two M/Erl/1 queues in series.

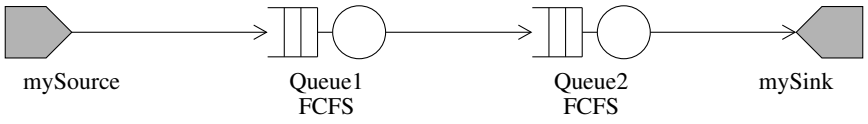

Figure 4.39: Topology of gallery_merl1_tandem: an open network over 2 queueing stations.

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.

```
public static Network gallery_merl1_tandem() {
    return gallery_merl1_linear(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.40 gallery_merlk

The M/Erl/k queue: Poisson arrivals, Erlang service, and k parallel servers.
The example is built and solved as follows.

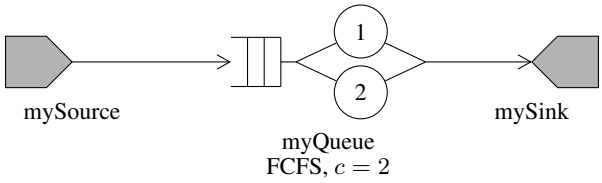

Figure 4.40: Topology of `gallery_merlk`: an open network over 1 queueing station.

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`.

```
public static Network gallery_merlk() {
    return gallery_merlk(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.41 `gallery_mhyp1`

The M/HyperExp/1 queue: Poisson arrivals, hyperexponential service, and a single server.

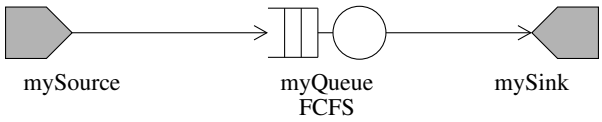

Figure 4.41: Topology of `gallery_mhyp1`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_mhyp1() {
    Network model = new Network("M/H/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, new Exp(1));
    queue.setService(oclass, HyperExp.fitMeanAndSCV(0.5, 4));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.
The example is built and solved as follows.

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`.

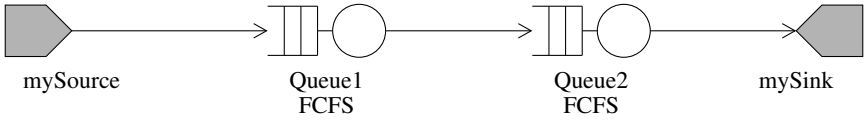

Figure 4.42: Topology of `gallery_mhyp1_linear`: an open network over 2 queueing stations.

```
public static Network gallery_mhyp1_linear() {
    return gallery_mhyp1_linear(2, 0.9);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.
The example is built and solved as follows.

```
public static Network gallery_mhyp1_reentrant() {
    Network model = new Network("M/Hyper/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, Exp.fitMean(1.00)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, Coxian.fitMeanAndSCV(0.5, 4)); // (Queue,Class1)
    node2.setService(jobclass2, Exp.fitMean(0.333333)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
```

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`.

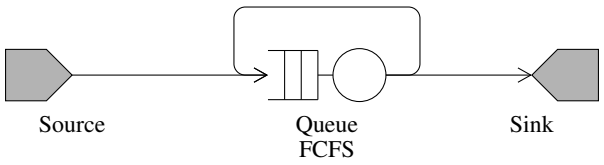

Figure 4.43: Topology of `gallery_mhyp1_reentrant`: an open network over 1 queueing station.

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`.

```
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.44 `gallery_mhyp1_tandem`

Two M/HyperExp/1 queues in series.

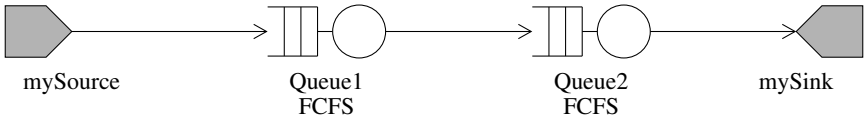

Figure 4.44: Topology of `gallery_mhyp1_tandem`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
public static Network gallery_mhyp1_tandem() {
    return gallery_mhyp1_linear(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.45 `gallery_mhypk`

The M/HyperExp/k queue: Poisson arrivals, hyperexponential service, and k parallel servers.
The example is built and solved as follows.

```
public static Network gallery_mhypk() {
    return gallery_mhypk(2);
```

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`.

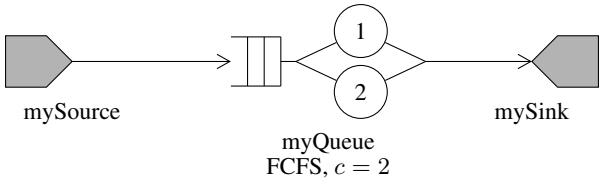

Figure 4.45: Topology of `gallery_mhypk`: an open network over 1 queueing station.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.46 gallery_mm1

The M/M/1 queue: Poisson arrivals, exponential service, and a single server.
The example is built and solved as follows.

```
Network model = new Network("M/M/1");
Source source = new Source(model, "mySource");
Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "mySink");
OpenClass oclass = new OpenClass(model, "myClass");
source.setArrival(oclass, new Exp(1));
queue.setService(oclass, new Exp(2));
model.link(Network.serialRouting(source, queue, sink));
return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

```
public static Network gallery_mm1_feedback() {
    return gallery_mm1_feedback(1.0 / 3);
```

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`.

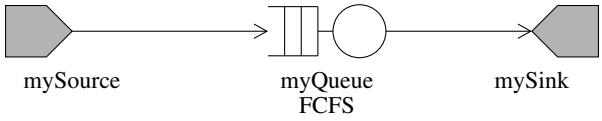

Figure 4.46: Topology of `gallery_mm1`: an open network over 1 queueing station.

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`.

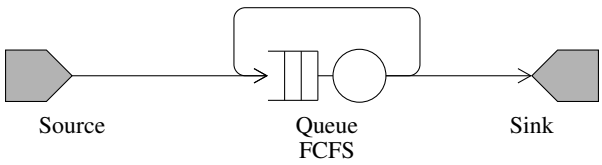

Figure 4.47: Topology of `gallery_mm1_feedback`: an open network over 1 queueing station.

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`.

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

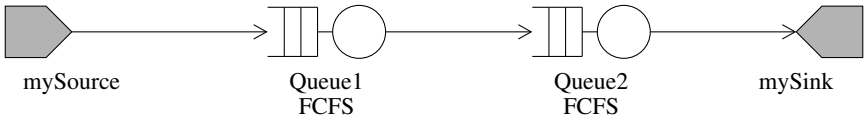

Figure 4.48: Topology of `gallery_mm1_linear`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
public static Network gallery_mm1_linear() {
    return gallery_mm1_linear(2, 0.9);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

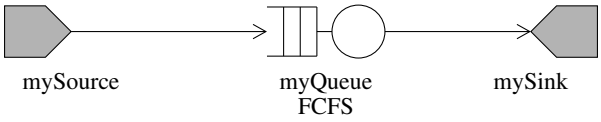

Figure 4.49: Topology of `gallery_mm1_multiclass`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_mm1_multiclass() {
    Network model = new Network("M[2]/M[2]/1");

    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");

    OpenClass oclass1 = new OpenClass(model, "myClass1");
    source.setArrival(oclass1, new Exp(1));
    queue.setService(oclass1, new Exp(4));
```

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`.

```
    OpenClass oclass2 = new OpenClass(model, "myClass2");
    source.setArrival(oclass2, new Exp(0.5));
    queue.setService(oclass2, new Exp(4));

    RoutingMatrix P = model.initRoutingMatrix();
    P.set(oclass1, Network.serialRouting(source, queue, sink));
    P.set(oclass2, Network.serialRouting(source, queue, sink));
    model.link(P);
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

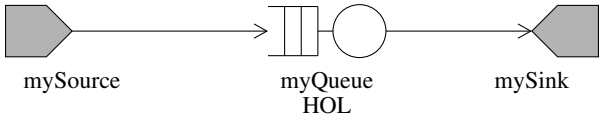

Figure 4.50: Topology of `gallery_mm1_prio`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1_prio() {
    Network model = new Network("M[2]/M[2]/1");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.HOL);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass1 = new OpenClass(model, "myClass1", 1);
    source.setArrival(oclass1, new Exp(1));
    queue.setService(oclass1, new Exp(4));
    OpenClass oclass2 = new OpenClass(model, "myClass2", 0);
    source.setArrival(oclass2, new Exp(0.5));
    queue.setService(oclass2, new Exp(4));
    RoutingMatrix P = model.initRoutingMatrix();
    P.set(oclass1, Network.serialRouting(source, queue, sink));
    P.set(oclass2, Network.serialRouting(source, queue, sink));
    model.link(P);
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

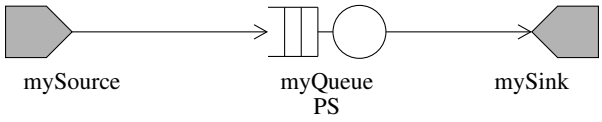

Figure 4.51: Topology of `gallery_mm1_ps`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1_ps() {
    Network model = new Network("M/M/1-PS");
    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.PS);
    Sink sink = new Sink(model, "mySink");
    OpenClass oclass = new OpenClass(model, "myClass");
    source.setArrival(oclass, new Exp(1));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

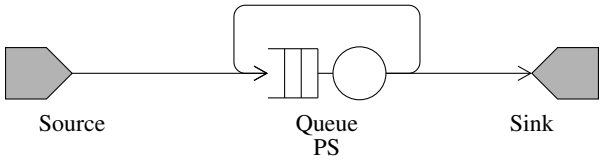

Figure 4.52: Topology of `gallery_mm1_ps_feedback`: an open network over 1 queueing station.

The example is built and solved as follows.

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`.

```
public static Network gallery_mm1_ps_feedback() {
    return gallery_mm1_ps_feedback(1.0 / 3.0);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

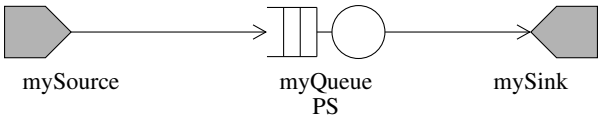

Figure 4.53: Topology of `gallery_mm1_ps_multiclass`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1_ps_multiclass() {
    Network model = new Network("M[2]/M[2]/1");

    Source source = new Source(model, "mySource");
    Queue queue = new Queue(model, "myQueue", SchedStrategy.PS);
    Sink sink = new Sink(model, "mySink");

    OpenClass oclass1 = new OpenClass(model, "myClass1");
    source.setArrival(oclass1, new Exp(1));
    queue.setService(oclass1, new Exp(4));

    OpenClass oclass2 = new OpenClass(model, "myClass2");
    source.setArrival(oclass2, new Exp(0.5));
    queue.setService(oclass2, new Exp(4));

    RoutingMatrix P = model.initRoutingMatrix();
    P.set(oclass1, Network.serialRouting(source, queue, sink));
    P.set(oclass2, Network.serialRouting(source, queue, sink));
    model.link(P);
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

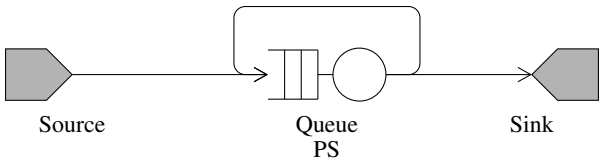

Figure 4.54: Topology of `gallery_mm1_ps_reentrant`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1_ps_reentrant() {
    Network model = new Network("M/M/1-PS-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.PS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, Exp.fitMean(1.00)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, Exp.fitMean(0.50)); // (Queue,Class1)
    node2.setService(jobclass2, Exp.fitMean(0.333333)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

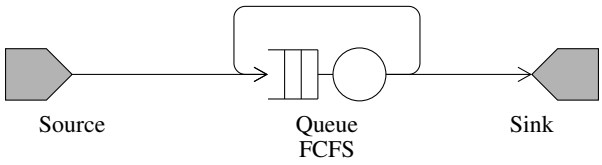

Figure 4.55: Topology of `gallery_mm1_reentrant`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1_reentrant() {
    Network model = new Network("M/M/1-Reentrant");

    // Block 1: nodes
    Source node1 = new Source(model, "Source");
    Queue node2 = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink node3 = new Sink(model, "Sink");

    // Block 2: classes
    OpenClass jobclass1 = new OpenClass(model, "Class1", 0);
    OpenClass jobclass2 = new OpenClass(model, "Class2", 0);

    node1.setArrival(jobclass1, Exp.fitMean(1.00)); // (Source,Class1)
    node1.setArrival(jobclass2, Disabled.getInstance()); // (Source,Class2)
    node2.setService(jobclass1, Exp.fitMean(0.50)); // (Queue,Class1)
    node2.setService(jobclass2, Exp.fitMean(0.333333)); // (Queue,Class2)

    // Block 3: topology
    RoutingMatrix routingMatrix = model.initRoutingMatrix();

    routingMatrix.set(jobclass1, jobclass1, node1, node2, 1.00); // (Source,Class1) -> (Queue
    ,Class1)
    routingMatrix.set(jobclass1, jobclass2, node2, node2, 1.00); // (CS_Queue_to_Queue,Class1
    ) -> (Queue,Class2)
    routingMatrix.set(jobclass2, jobclass2, node2, node3, 1.00); // (Queue,Class2) -> (Sink,
    Class2)

    model.link(routingMatrix);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.56 gallery_mm1_tandem

Two M/M/1 queues in series.

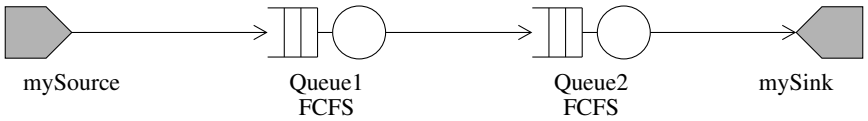

Figure 4.56: Topology of `gallery_mm1_tandem`: an open network over 2 queueing stations.

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`.

The example is built and solved as follows.

```
public static Network gallery_mm1_tandem() {
    return gallery_mm1_linear(2, 0.9);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.57 gallery_mm1_tandem_multiclass

Two M/M/1 queues in series under a multiclass workload.

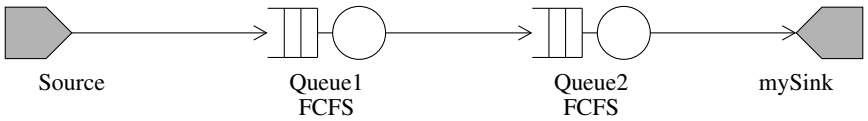

Figure 4.57: Topology of `gallery_mm1_tandem_multiclass`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
public static Network gallery_mm1_tandem_multiclass() {
    Network model = new Network("M[2]/M[2]/1 -> -/M[2]/1");

    Source source = new Source(model, "Source");
    Queue queue1 = new Queue(model, "Queue1", SchedStrategy.FCFS);
    Queue queue2 = new Queue(model, "Queue2", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "mySink");

    OpenClass oclass1 = new OpenClass(model, "myClass1");
    source.setArrival(oclass1, new Exp(1));
    queue1.setService(oclass1, new Exp(4));
    queue2.setService(oclass1, new Exp(6));

    OpenClass oclass2 = new OpenClass(model, "myClass2");
    source.setArrival(oclass2, new Exp(0.5));
    queue1.setService(oclass2, new Exp(2));
    queue2.setService(oclass2, new Exp(6));

    RoutingMatrix P = model.initRoutingMatrix();
```

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`.

```
    P.set(oclass1, Network.serialRouting(source, queue1, queue2, sink));
    P.set(oclass2, Network.serialRouting(source, queue1, queue2, sink));
    model.link(P);

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

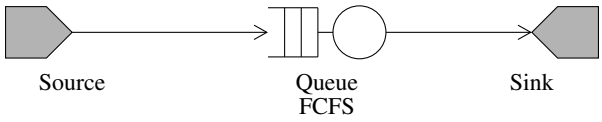

Figure 4.58: Topology of `gallery_mm1k`: an open network over 1 queueing station.

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.

```
public static Network gallery_mm1k() {
    return gallery_mm1k(3);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.59 `gallery_mmap1`

The M/MAP/1 queue: Poisson arrivals, MAP-modulated service, and a single server.
The example is built and solved as follows.

```
public static Network gallery_mmap1() {
    MAP map = MAP.rand();
    map.setMean(0.5);
```

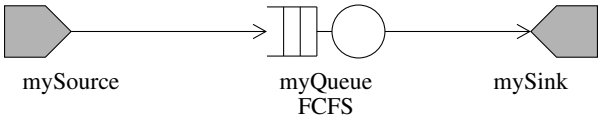

Figure 4.59: Topology of `gallery_mmap1`: an open network over 1 queueing station.

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`.

```
    return gallery_mmap1(map);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

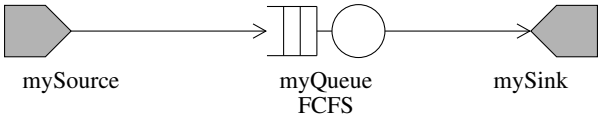

Figure 4.60: Topology of `gallery_mmap1_multiclass`: an open network over 1 queueing station.

The example is built and solved as follows.

```
public static Network gallery_mmap1_multiclass() {
    MAP map1 = MAP.rand();
    map1.setMean(0.5);
    MAP map2 = MAP.rand();
    map2.setMean(0.5);
    return gallery_mmap1_multiclass(map1, map2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.61 `gallery_mmapk`

The M/MAP/k queue: Poisson arrivals, MAP-modulated service, and k parallel servers.
The example is built and solved as follows.

```
public static Network gallery_mmapk() {
    return gallery_mmapk(MAP.rand(), 2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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`.

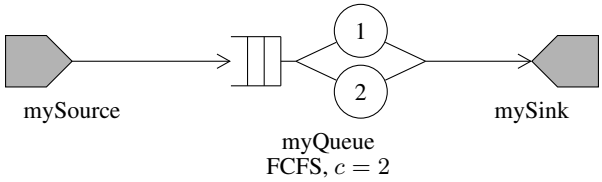

Figure 4.61: Topology of `gallery_mmapk`: an open network over 1 queueing station.

4.2.62 `gallery_mmk`

The M/M/k queue: Poisson arrivals, exponential service, and k parallel servers.
The example is built and solved as follows.

```
public static Network gallery_mmk() {
    return gallery_mmk(2);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.63 `gallery_mpar1`

The M/Pareto/1 queue: Poisson arrivals, Pareto service, and a single server.
The example is built and solved as follows.

```
public static Network gallery_mpar1() {
    Network model = new Network("M/Par/1");
    Source source = new Source(model, "Source");
    Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass = new OpenClass(model, "Class1");
    source.setArrival(oclass, new Exp(1));
    queue.setService(oclass, Pareto.fitMeanAndSCV(0.5, 64));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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`.

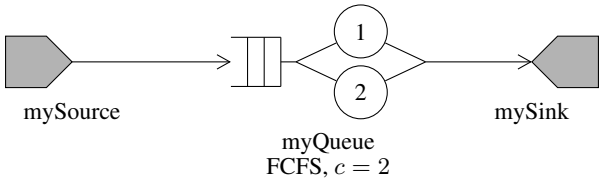

Figure 4.62: Topology of `gallery_mmk`: an open network over 1 queueing station.

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`.

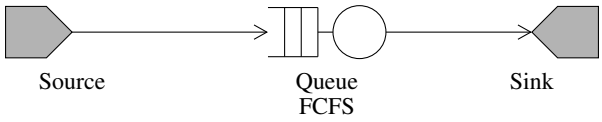

Figure 4.63: Topology of `gallery_mpar1`: an open network over 1 queueing station.

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`.

4.2.64 gallery_multitier

A multitier application: presentation, application and database tiers as a closed network.

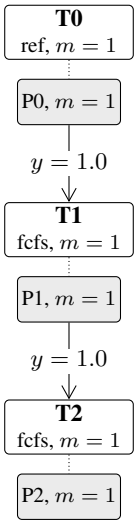

Figure 4.64: Call graph of gallery_multitier: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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.

```
public static LayeredNetwork gallery_multitier() {
    LayeredNetwork model = new LayeredNetwork("testLQN3");

    // Layer 1: client
    Processor P0 = new Processor(model, "P0", 1, SchedStrategy.PS);
    Task T0 = new Task(model, "T0", 1, SchedStrategy.REF).on(P0);
    Entry E0 = new Entry(model, "E0").on(T0);

    // Layer 2: application server
    Processor P1 = new Processor(model, "P1", 1, SchedStrategy.PS);
    Task T1 = new Task(model, "T1", 1, SchedStrategy.FCFS).on(P1);
    Entry E10 = new Entry(model, "E10").on(T1);
    Entry E11 = new Entry(model, "E11").on(T1);
    Entry E12 = new Entry(model, "E12").on(T1);
    Entry E13 = new Entry(model, "E13").on(T1);

    // Layer 3: database
    Processor P2 = new Processor(model, "P2", 1, SchedStrategy.PS);
    Task T2 = new Task(model, "T2", 1, SchedStrategy.FCFS).on(P2);
    Entry E20 = new Entry(model, "E20").on(T2);
    Entry E21 = new Entry(model, "E21").on(T2);
    Entry E22 = new Entry(model, "E22").on(T2);
    Entry E23 = new Entry(model, "E23").on(T2);

    // Client activities
    Activity A0 = new Activity(model, "A0", new Exp(1.0)).on(T0).boundTo(E0).synchCall(E12,
    1.0);
```

```
    ... (40 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    b7sources.add(B7b);
    T1.addPrecedence(ActivityPrecedence.OrJoin(b7sources, B8));
    T2.addPrecedence(ActivityPrecedence.Serial(C0, C1));
    T2.addPrecedence(ActivityPrecedence.Serial(C3, C4, C5));

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.65 `gallery_multitier_storage`

The multitier application with an explicit storage tier behind the database.

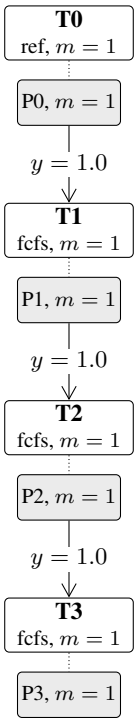

Figure 4.65: Call graph of `gallery_multitier_storage`: 4 tasks over 4 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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`.

The example is built and solved as follows.

```
public static LayeredNetwork gallery_multitier_storage() {
    LayeredNetwork model = new LayeredNetwork("testLQN3_Cache");
```

```
    // Layer 1: client
    Processor P0 = new Processor(model, "P0", 1, SchedStrategy.PS);
    Task T0 = new Task(model, "T0", 1, SchedStrategy.REF).on(P0);
    Entry E0 = new Entry(model, "E0").on(T0);

    // Layer 2: application server
    Processor P1 = new Processor(model, "P1", 1, SchedStrategy.PS);
    Task T1 = new Task(model, "T1", 1, SchedStrategy.FCFS).on(P1);
    Entry E10 = new Entry(model, "E10").on(T1);
    Entry E11 = new Entry(model, "E11").on(T1);
    Entry E12 = new Entry(model, "E12").on(T1);
    Entry E13 = new Entry(model, "E13").on(T1);

    // Layer 3: database
    Processor P2 = new Processor(model, "P2", 1, SchedStrategy.PS);
    Task T2 = new Task(model, "T2", 1, SchedStrategy.FCFS).on(P2);
    Entry E20 = new Entry(model, "E20").on(T2);
    Entry E21 = new Entry(model, "E21").on(T2);
    Entry E22 = new Entry(model, "E22").on(T2);
    Entry E23 = new Entry(model, "E23").on(T2);

    // Layer 4: cache (10 items, capacity 2, LRU, uniform access)
    int totalitems = 10;

    ... (57 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

    List<Activity> cacheTargets = new ArrayList<Activity>();
    cacheTargets.add(D1a);
    cacheTargets.add(D1b);
    T3.addPrecedence(ActivityPrecedence.CacheAccess(D0, cacheTargets));

    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.66 `gallery_parm1`

The Pareto/M/1 queue: Pareto interarrival times, exponential service, and a single server.

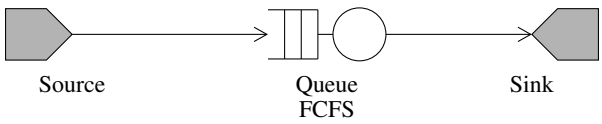

Figure 4.66: Topology of `gallery_parm1`: an open network over 1 queueing station.

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.

```
public static Network gallery_parm1() {
    Network model = new Network("Par/M/1");
    Source source = new Source(model, "Source");
    Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
```

```
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass = new OpenClass(model, "Class1");
    source.setArrival(oclass, Pareto.fitMeanAndSCV(1, 64));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.67 `gallery_qn_random`

A randomly generated queueing network, used as a generator-driven parity fixture.

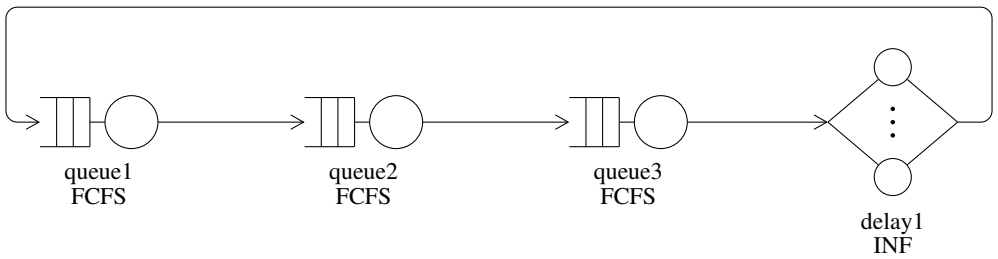

Figure 4.67: Topology of `gallery_qn_random`: a closed network over 4 queueing stations.

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.

```
public static Network gallery_qn_random() {
    return gallery_qn_random(23000L);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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.

```
public static Environment gallery_renv_breakdown() {
    Network model = new Network("ServerWithFailures");
    Source source = new Source(model, "Arrivals");
    Queue queue = new Queue(model, "Server", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Departures");
    OpenClass jobclass = new OpenClass(model, "Jobs");
    source.setArrival(jobclass, new Exp(0.8));
```

```
    queue.setService(jobclass, new Exp(2.0));
    queue.setNumberOfServers(1);
    RoutingMatrix P = model.initRoutingMatrix();
    P.set(jobclass, jobclass, source, queue, 1.0);
    P.set(jobclass, jobclass, queue, sink, 1.0);
    model.link(P);
    Environment env = new Environment("ServerEnv");
    env.addNodeFailureRepair(model, queue, new Exp(0.1), new Exp(1.0), new Exp(0.5));
    env.init();
    return env;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.69 gallery_repairmen

The machine repairman model: machines think, fail, and queue for one repair station.

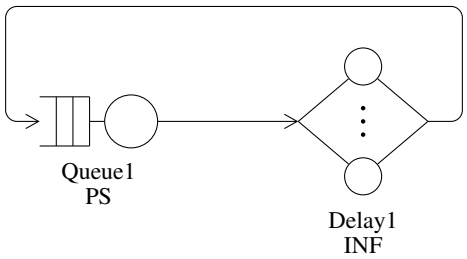

Figure 4.68: Topology of gallery_repairmen: a closed network over 2 queueing stations.

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.

```
public static Network gallery_repairmen() {
    return gallery_repairmen(2, 23000);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.70 gallery_replayerm1

A single queue whose interarrival times are replayed from a measurement trace.

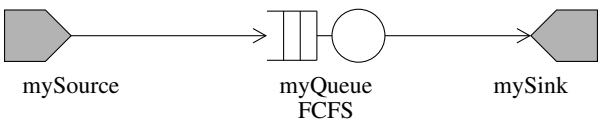

Figure 4.69: Topology of gallery_replayerm1: an open network over 1 queueing station.

The example is built and solved as follows.

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`.

```
public static Network gallery_replayerm1() {
    URI fileURI = null;
    try {
        fileURI = Gallery.class.getResource("/example_trace.txt").toURI();
    } catch (Exception e) {
        e.printStackTrace();
    }
    assert fileURI != null;
    String fileName = Paths.get(fileURI).toString();
    return gallery_replayerm1(fileName);
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

4.2.71 `gallery_um1`

The U/M/1 queue: uniform interarrival times, exponential service, and a single server.

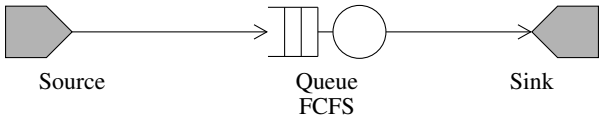

Figure 4.70: Topology of `gallery_um1`: an open network over 1 queueing station.

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.

```
public static Network gallery_um1() {
    Network model = new Network("U/M/1");
    Source source = new Source(model, "Source");
    Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
    Sink sink = new Sink(model, "Sink");
    OpenClass oclass = new OpenClass(model, "Class1");
    source.setArrival(oclass, new Uniform(1, 2));
    queue.setService(oclass, new Exp(2));
    model.link(Network.serialRouting(source, queue, sink));
    return model;
```

Running this edition prints

```
no such class: jline.examples.java.Gallery
```

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 `jline.examples.java.discrete`, and slotted mode is requested through the solver options rather than by the model.

```
Network model = new Network("GeoGeo1");
Source source = new Source(model, "Source");
Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Sink");
OpenClass jobClass = new OpenClass(model, "Class1");
source.setArrival(jobClass, new Geometric(a));
queue.setService(jobClass, new Geometric(s));
model.link(Network.serialRouting(source, queue, sink));

new SolverNC(model, slotted()).getAvgTable().print();
System.out.printf("closed form: E[N] = %g, E[T] = %g slots%n",
        a * (1 - a) / (s - a), (1 - a) / (s - a));
```

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.

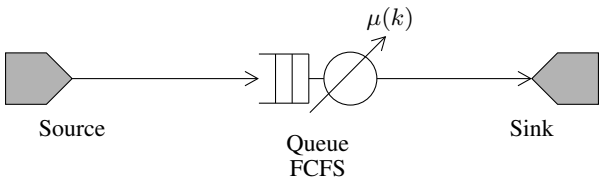

Figure 5.1: Topology of `dt_bernoulli_loaddep`: an open network over 1 queueing station.

The example is built and solved as follows.

```
System.out.println("\n--- ex3_bernoulli_loaddep: example 2.10 and theorem 2.11 ---");
double a = 0.6;
double s = 0.3;
```

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`.

```
int c = 3;
int L = 20;
Network model = DiscreteModel.dt_bernoulli_loaddep(a, s, c, L);
new SolverNC(model, slotted()).getAvgTable().print();
double[] p = new double[L];
for (int n = 1; n <= L; n++) {
    p[n - 1] = s * Math.min(n, c);
}
Bernoulli1Result r = Dqsys_bernoulli1.dqsys_bernoulli1(new double[]{a}, p, L);
System.out.println("time-stationary law pi(0..6)  : "
        + Arrays.toString(Arrays.copyOf(r.pmf, 7)));
System.out.println("arrival law         pi_1(0..6): "
        + Arrays.toString(Arrays.copyOf(r.arrivalPmf, 7)));
System.out.println("no PASTA in discrete time: the two rows above are different laws");
```

Running this edition prints

```
--- ex3_bernoulli_loaddep: example 2.10 and theorem 2.11 ---
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: java; 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)  : [0.045977011494253 46, 0.22988505747126736, 0.40229885057471787,
    0.2681992337164785, 0.04469987228607978, 0.007449978714346 6325, 0.001241663119057774]
arrival law         pi_1(0..6): [0.11494252873563435, 0.40229885057472026, 0.40229885057472026,
    0.06704980842912008, 0.011174968071520018, 0.0018624946785866705, 3.104157797644452E-4]
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.

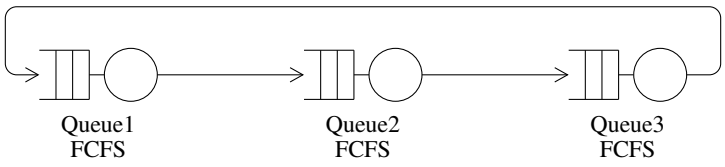

Figure 5.2: Topology of `dt_cycle`: a closed network over 3 queueing stations.

The example is built and solved as follows.

```
System.out.println("\n--- ex4_cycle: closed cycle, corollary 3.4 ---");
double[] p = DiscreteModel.defaultCycleRates();
int N = 5;
Network model = DiscreteModel.dt_cycle(p, N);
new SolverNC(model, slotted()).getAvgTable().print();
DpfqnNcResult nc = Dpfqn_nc.dpfqn_nc(p, N);
System.out.printf("log G(N,J)    = %g%n", nc.lG);
System.out.printf("throughput    = %g jobs per slot%n", nc.throughput());
```

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`.

```
double[] util = new double[p.length];
for (int j = 0; j < p.length; j++) {
    util[j] = nc.throughput() / p[j];
}
System.out.println("utilizations = " + Arrays.toString(util));
```

Running this edition prints

```
--- ex4_cycle: closed cycle, corollary 3.4 ---
NC analysis [method: dt.cycle; type: exact, deterministic; lang: java; 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 = [0.4965639088467044, 0.9931278176934089, 0.3546885063190746]
```

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.

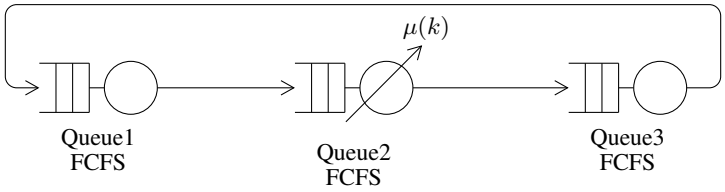

Figure 5.3: Topology of `dt_cycle_loaddep`: a closed network over 3 queueing stations.

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`.

The example is built and solved as follows.

```
System.out.println("\n--- ex5_cycle_loaddep: closed cycle, theorem 3.2 ---");
double[] p = DiscreteModel.defaultCycleRates();
int N = 5;
Network model = DiscreteModel.dt_cycle_loaddep(p, N, 1, 2);
new SolverNC(model, slotted()).getAvgTable().print();
```

```
double[][] P = new double[p.length][N];
for (int j = 0; j < p.length; j++) {
    for (int n = 1; n <= N; n++) {
        P[j][n - 1] = (j == 1) ? p[j] * Math.min(n, 2) : p[j];
    }
}
DpfqnNcLdResult nc = Dpfqn_ncld.dpfqn_ncld(P, N);
System.out.println("P(X_2 = 0..N) = " + Arrays.toString(nc.marginal(1)));
```

Running this edition prints

```
--- ex5_cycle_loaddep: closed cycle, theorem 3.2 ---
NC analysis [method: dt.cycleld; type: exact, deterministic; lang: java; 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..N) = [0.05243257509316642, 0.20483277208263767, 0.29010780910885847, 0.25011132807362985,
    0.15678620565809628, 0.045729309983611396]
```

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.

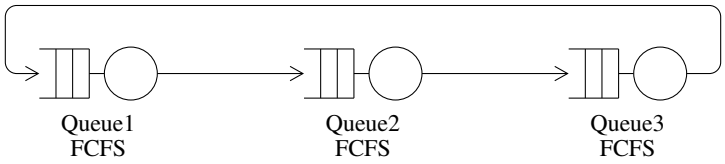

Figure 5.4: Topology of `dt_cycle_multiclass`: a closed network over 3 queueing stations.

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`.

The example is built and solved as follows.

```
System.out.println("\n--- ex6_cycle_multiclass: multichain cycle, section 3.2 ---");
double[] p = DiscreteModel.defaultCycleRates();
int[] pops = new int[]{3, 2};
Network model = DiscreteModel.dt_cycle_multiclass(p, pops);
new SolverNC(model, slotted()).getAvgTable().print();
DpfqnNcResult nc = Dpfqn_nc.dpfqn_nc(p, pops[0] + pops[1]);
double X = nc.throughput();
System.out.printf("aggregate throughput  = %g jobs per slot%n", X);
System.out.printf("per-chain throughput  = %g and %g%n",
        X * pops[0] / (pops[0] + pops[1]), X * pops[1] / (pops[0] + pops[1]));
```

Running this edition prints

```
--- ex6_cycle_multiclass: multichain cycle, section 3.2 ---
NC analysis [method: dt.cycle; type: exact, deterministic; lang: java; 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.

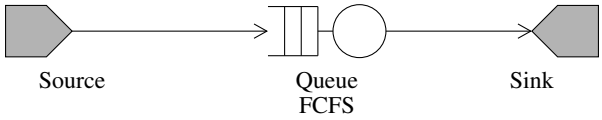

Figure 5.5: Topology of `dt_geogeo1`: an open network over 1 queueing station.

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.

```
System.out.println("\n--- ex1_geogeo1: Geo/Geo/1 on the slot lattice ---");
double a = 0.2;
double s = 0.5;
Network model = DiscreteModel.dt_geogeo1(a, s);
new SolverNC(model, slotted()).getAvgTable().print();
System.out.printf("closed form: E[N] = %g, E[T] = %g slots%n",
        a * (1 - a) / (s - a), (1 - a) / (s - a));
```

Running this edition prints

```
--- ex1_geogeo1: Geo/Geo/1 on the slot lattice ---
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: java; 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
```

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.

```
System.out.println("\n--- ex2_geogeo1_loss: finite buffer, corollary 2.8 ---");
double a = 0.2;
double s = 0.5;
```

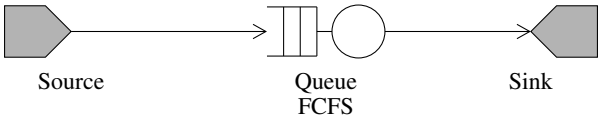

Figure 5.6: Topology of `dt_geogeo1_loss`: an open network over 1 queueing station.

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`.

```
int L = 4;
Network model = DiscreteModel.dt_geogeo1_loss(a, s, L);
new SolverNC(model, slotted()).getAvgTable().print();
Bernoulli1Result r = Dqsys_bernoulli1.dqsys_bernoulli1(new double[]{a}, new double[]{s}, L);
System.out.println("queue length law   : " + Arrays.toString(r.pmf));
System.out.printf("loss probability   : %g%n", r.lossProb);
System.out.printf("carried throughput : %g of the %g offered per slot%n", r.throughput, a);
```

Running this edition prints

```
--- ex2_geogeo1_loss: finite buffer, corollary 2.8 ---
NC analysis [method: dt.bernoulli1; type: exact, deterministic; lang: java; 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   : [0.6015037593984962, 0.3007518796992481, 0.07518796992481204, 0.018796992481203006,
     0.0037593984962406026]
loss probability   : 0.00375940
carried throughput : 0.199248 of the 0.200000 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 the solver example classes.

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.

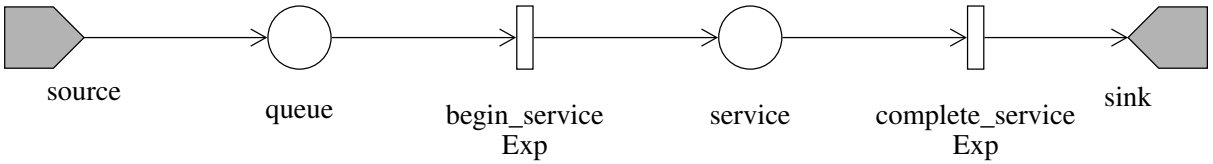

Figure 6.1: Topology of `ctmc_spn_mm1`: a stochastic Petri net over 2 places and 2 transitions; a place is a circle holding its initial marking and a transition a bar, hollow when its firing time is random and solid when it fires at once; the net is open, fed by a Source and drained into a Sink.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

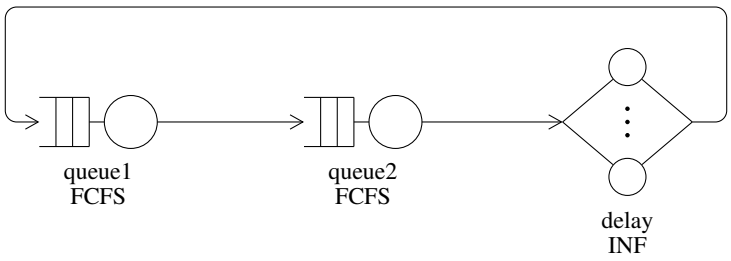

Figure 6.2: Topology of `ctmc_spn_vs_nc_closed_qn`: a closed network over 3 queueing stations.

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`.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

6.0.3 `ctmc_tandem_mm1`

The generator of a two-station tandem, its state space and its stationary distribution, printed in full.

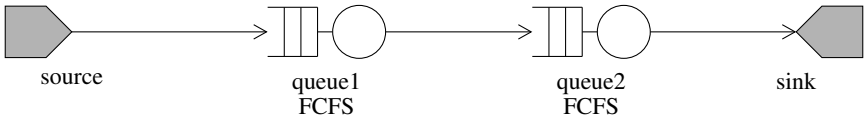

Figure 6.3: Topology of `ctmc_tandem_mm1`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
System.out.println("================================================================================"
    );
System.out.println("Tandem M/M/1 Queues (Series of Two Stations)");
System.out.println("================================================================================"
    );

Network model = new Network("tandem_mm1");
Source source = new Source(model, "source");
Queue queue1 = new Queue(model, "queue1", SchedStrategy.FCFS);
Queue queue2 = new Queue(model, "queue2", SchedStrategy.FCFS);
Sink sink = new Sink(model, "sink");
OpenClass jobclass = new OpenClass(model, "jobs");

source.setArrival(jobclass, Exp.fitMean(1 / 0.6));
queue1.setService(jobclass, Exp.fitMean(1.0));
queue2.setService(jobclass, Exp.fitMean(1 / 1.2));

RoutingMatrix R = model.initRoutingMatrix();
R.set(jobclass, jobclass, source, queue1, 1.0);
R.set(jobclass, jobclass, queue1, queue2, 1.0);
R.set(jobclass, jobclass, queue2, sink, 1.0);
model.link(R);

System.out.println("\nSolving tandem M/M/1 system with CTMC...");
System.out.println("Parameters:");
```

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`.

```
System.out.println("  Station 1: lambda=0.6, mu=1.0, rho=0.6");
System.out.println("  Station 2: lambda=0.6, mu=1.2, rho=0.5\n");


    ... (3 source lines omitted; full implementation: jar/src/main/java/jline/examples/)


System.out.println("\n
    ========================================================================");
System.out.println("CTMC Results Summary:");
System.out.println("========================================================================"
    );
report(avgTable, "queue1", "Queue 1");
report(avgTable, "queue2", "Queue 2");
```

No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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 the reader and writer tests 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. The JSON examples have no counterpart in the `jline.examples` tree: the Java side exercises the format through its reader and writer tests instead.

```
Network model = JsonModel.heteroServers();
LineModelIO.save(model, "model.json");
Network reloaded = LineModelIO.load("model.json");
```

7.1 The fixtures

7.1.1 `json_balking_retrial`

Balking thresholds and retrial orbits over the wire.

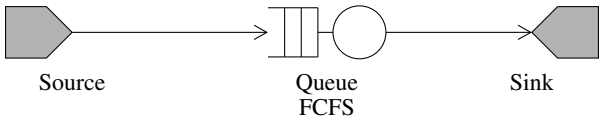

Figure 7.1: Topology of `json_balking_retrial`: an open network over 1 queueing station.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

7.1.2 `json_classcap_droprule`

Per-class capacities and drop rules.

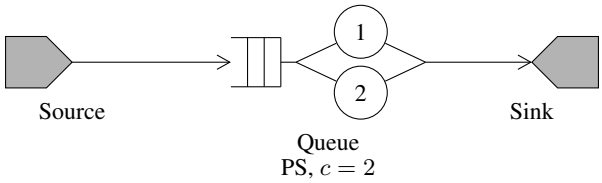

Figure 7.2: Topology of `json_classcap_droprule`: an open network over 1 queueing station.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

7.1.3 `json_fcr_details`

The finite capacity region block, with its members, limits and drop rules.

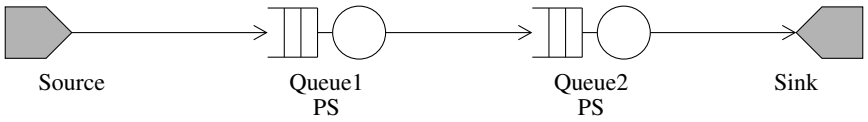

Figure 7.3: Topology of `json_fcr_details`: an open network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

7.1.4 `json_hetero_servers`

Heterogeneous server types at one station, their per-type service rates and the policy that orders them.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

7.1.5 `json_join_deadline`

Join deadlines and the quorum they imply.

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`.

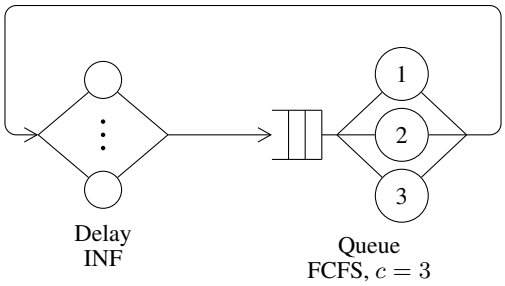

Figure 7.4: Topology of `json_hetero_servers`: a closed network over 2 queueing stations.

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`.

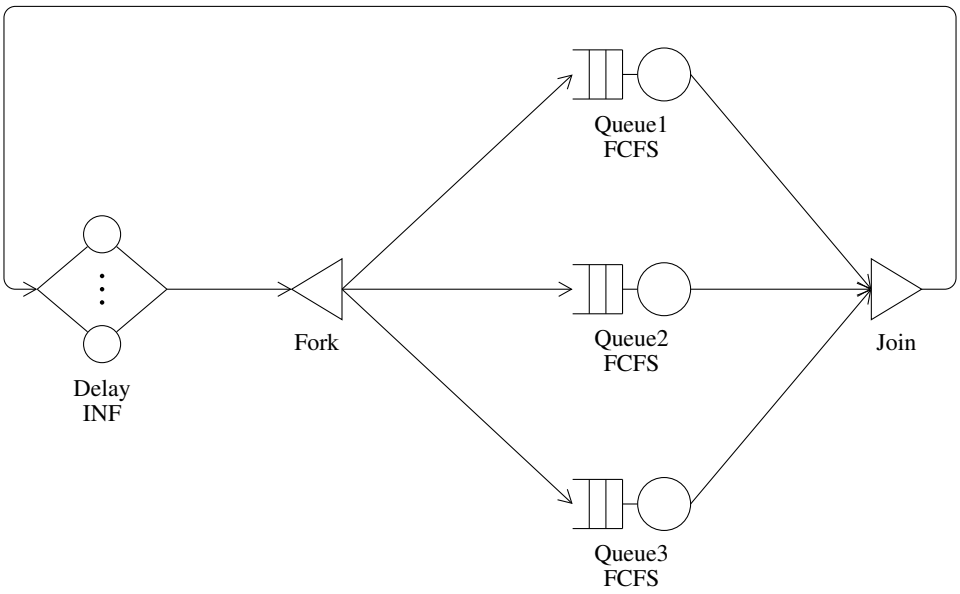

Figure 7.5: Topology of `json_join_deadline`: a closed network over 4 queueing stations with a fork-join block.

Nodes	6: Queue × 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`.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

7.1.6 `json_signal_classes`

G-network signal classes.

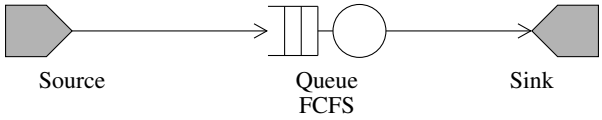

Figure 7.6: Topology of `json_signal_classes`: an open network over 1 queueing station.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. Of the inference examples only the variational one has a counterpart in the `jline.examples` tree; the Java side exercises the other estimators through the inference API and its tests.

The scripts live under `jline.examples.java.inference` and the inference API, 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
queue.setService(jobclass1, new Exp(Double.NaN));   // true demand 0.1
queue.setService(jobclass2, new Exp(Double.NaN));   // true demand 0.3
```

8.1 The estimators

8.1.1 `est_ekf_changepoint`

The filter under a changepoint in the workload.

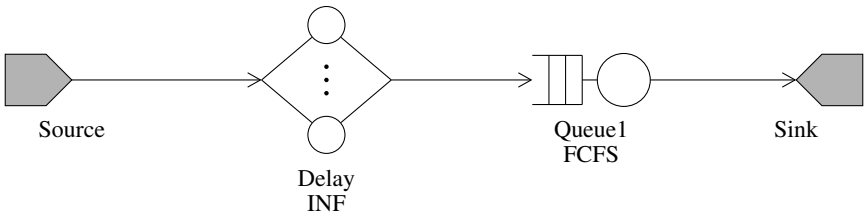

Figure 8.1: Topology of `est_ekf_changepoint`: an open network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

8.1.2 `est_ekf_closed`

The closed-network filter.

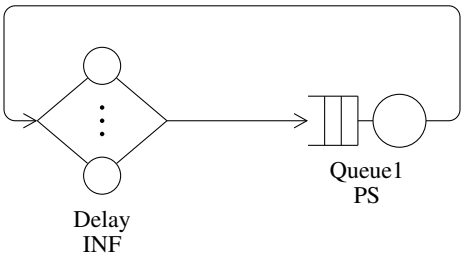

Figure 8.2: Topology of `est_ekf_closed`: a closed network over 2 queueing stations.

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`.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.3 `est_ekf_open`

An extended Kalman filter tracking the demands of an open network online.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.4 `est_erps_closed`

The closed-network smoother.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

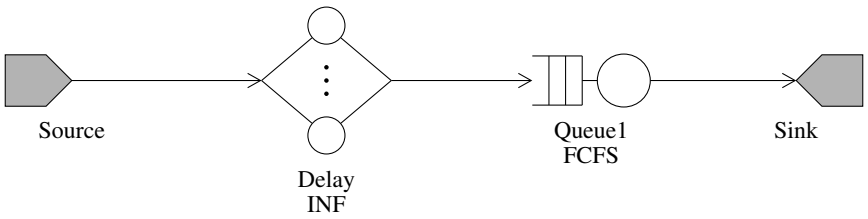

Figure 8.3: Topology of `est_ekf_open`: an open network over 2 queueing stations.

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`.

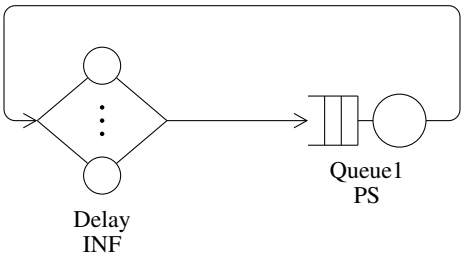

Figure 8.4: Topology of `est_erps_closed`: a closed network over 2 queueing stations.

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`.

8.1.5 `est_erps_open`

The Rauch-Tung-Striebel smoother on an open network.

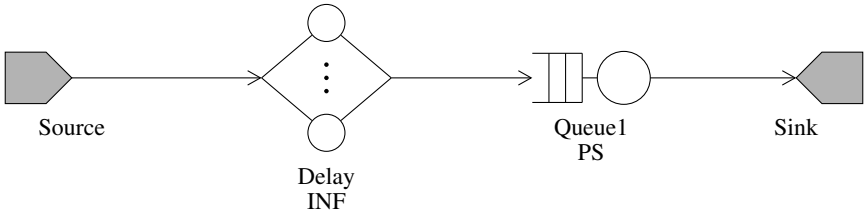

Figure 8.5: Topology of `est_erps_open`: an open network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.6 `est_mcmc_closed`

Markov chain Monte Carlo sampling of the posterior over the demands.

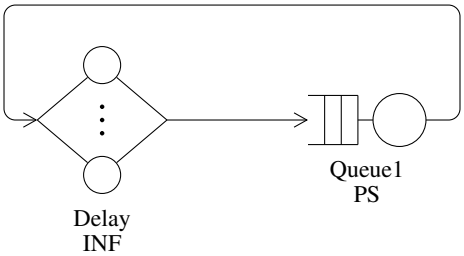

Figure 8.6: Topology of `est_mcmc_closed`: a closed network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository

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`.

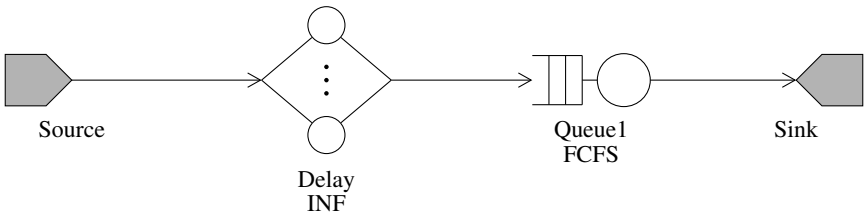

Figure 8.7: Topology of `est_mle_open`: an open network over 2 queueing stations.

for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.8 `est_qmle_closed`

Quasi-likelihood estimation on a closed network.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.10 `est_ubo_changepoint`

The variational estimator under a changepoint.

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`.

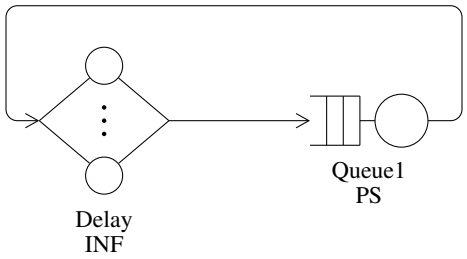

Figure 8.8: Topology of `est_qmle_closed`: a closed network over 2 queueing stations.

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`.

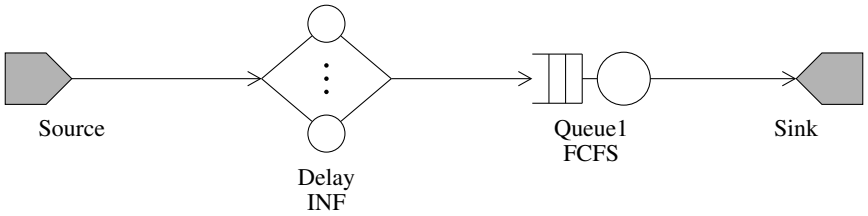

Figure 8.9: Topology of `est_ubo_changepoint`: an open network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.11 `est_ubo_closed`

The closed-network version.

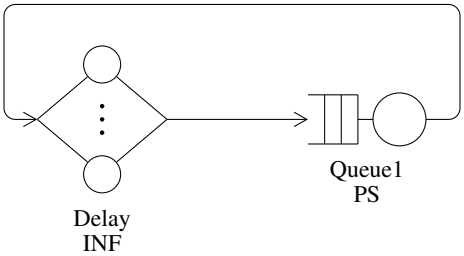

Figure 8.10: Topology of `est_ubo_closed`: a closed network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.12 `est_ubo_open`

Variational Bayesian estimation on an open network.

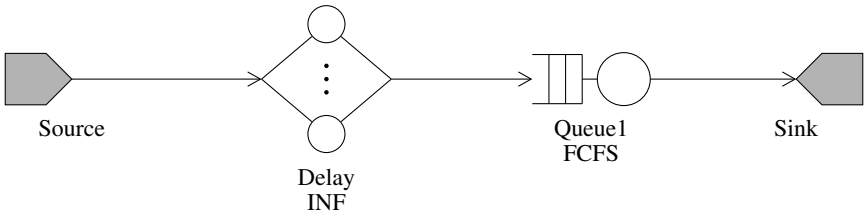

Figure 8.11: Topology of `est_ubo_open`: an open network over 2 queueing stations.

The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

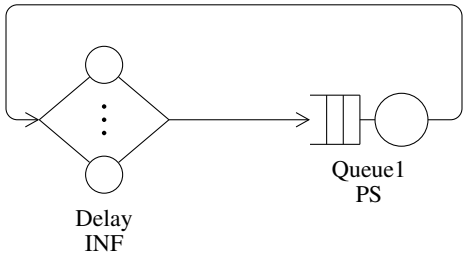

Figure 8.12: Topology of `est_ubo_variants_closed`: a closed network over 2 queueing stations.

8.1.13 `est_ubo_variants_closed`

The variants of the variational estimator, side by side.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.14 `est_ubr_changepoint`

The regression estimator under a changepoint.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

8.1.15 `est_ubr_closed`

The closed-network version.
The example is built and solved as follows. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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`.

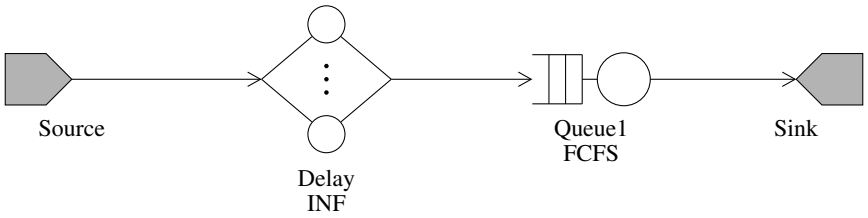

Figure 8.13: Topology of `est_ubr_changepoint`: an open network over 2 queueing stations.

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`.

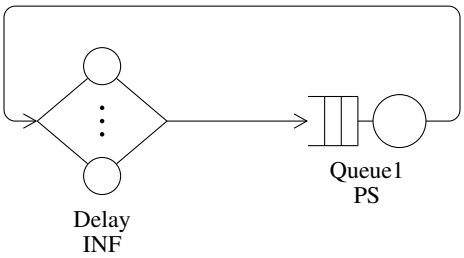

Figure 8.14: Topology of `est_ubr_closed`: a closed network over 2 queueing stations.

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`.

8.1.16 est_ubr_open

The regression-based estimator on an open network.

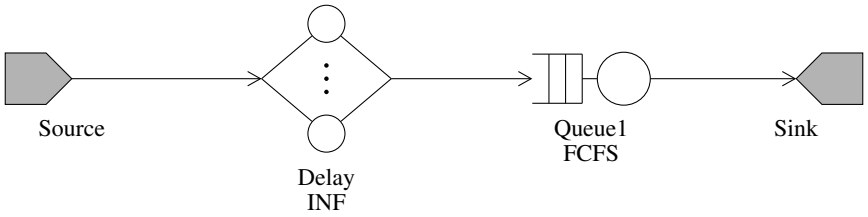

Figure 8.15: Topology of `est_ubr_open`: an open network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

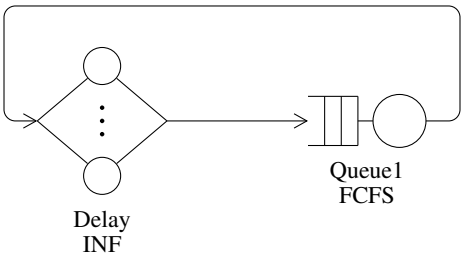

Figure 8.16: Topology of `est_vi_closed`: a closed network over 2 queueing stations.

The example is built and solved as follows.

```
int N = 10;

// define model, with the queue rate to be estimated
Network model = new Network("model");
Delay delay = new Delay(model, "Delay");
Queue queue = new Queue(model, "Queue1", SchedStrategy.FCFS);
ClosedClass jobclass = new ClosedClass(model, "Class1", N, delay, 0);
delay.setService(jobclass, new Exp(0.5));
queue.setService(jobclass, new Exp(2.0));   // starting point of the estimate
model.link(model.serialRouting(delay, queue));

// queue-length readings, one per unit time, 10% of them faulty
```

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`.

```
double[] qlen = {1, 2, 3, 2, 4, 3, 5, 4, 3, 4};
double[] ts = new double[qlen.length];
double[] dlen = new double[qlen.length];
for (int k = 0; k < qlen.length; k++) {
    ts[k] = k + 1;
    dlen[k] = N - qlen[k];
}

// estimate the service rate of the queue
ParamEstimator se = new ParamEstimator(model);
se.options.method = "vi";
se.options.epsilon = 0.1;      // probability that a reading is faulty
se.options.priorShape = 2.0;   // Gamma prior shape; the rate is set from the model
se.options.variational.ngrid = 51;      // time grid of the backward and forward passes

    ... (14 source lines omitted; full implementation: jar/src/main/java/jline/examples/)

}
System.out.println();

// solve the model the estimate has been written into
System.out.println("SOLVER: MVA");
new SolverMVA(model).getAvgTable().print();
```

Running this edition prints

```
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 -174.650 -163.234
SOLVER: MVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; 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 `jline.examples.opt.LineOptExamples` class 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

Cost coefficients are supplied as `Map<Station,Double>` instances, and solver options are collected in a `LineOptSolverOptions` object whose setters return the object itself, allowing chained configuration.

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 (`LineOptExamples.serverSizing`) solves the problem with the default differential evolution solver, at a cost of 10 units per server:

```
// Block 1: LINE model (Source -> Queue -> Sink)
Network model = new Network("MMc");
Source source = new Source(model, "Arrivals");
Queue queue = new Queue(model, "Server", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Departures");
OpenClass jobs = new OpenClass(model, "Jobs");
source.setArrival(jobs, new Exp(3.0));    // lambda = 3
queue.setService(jobs, new Exp(1.0));     // mu = 1 per server
model.link(Network.serialRouting(source, queue, sink));

// Block 2: decision variables
OptimizationProblem p = new OptimizationProblem(model);
```

```
p.addVariable(new ServerAllocation(queue, 1, 10));

// Block 3: objective and constraints
Map<Station, Double> serverCost = new LinkedHashMap<Station, Double>();
serverCost.put(queue, 10.0);
p.setObjective(new MinimizeCost(serverCost, null, null, null));
p.addConstraint(new UtilizationConstraint(queue, 0.5));

// Block 4: solution
OptimizationResult r = p.solve(new LineOptSolverOptions().setSeed(42));
System.out.println("servers=" + r.getVariableValue("Server_servers")
        + " cost=" + r.objectiveValue + " evals=" + r.modelEvaluations);
```

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 (`LineOptExamples.bisectionSizing`); only the solution block changes:

```
OptimizationResult r = new BisectionSolver(p).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 (`LineOptExamples.serviceRate`), the model is an M/M/1 queue built as in Tutorial 1 with initial rate $\mu = 4$, and the problem is declared as:

```
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServiceRate(queue, jobs, 3.5, 8.0));
Map<Station, Double> rateCost = new LinkedHashMap<Station, Double>();
rateCost.put(queue, 20.0);
p.setObjective(new MinimizeCost(null, rateCost, null, null));
p.addConstraint(new ResponseTimeConstraint(queue, jobs, 0.5));

OptimizationResult r = p.solve(
        new LineOptSolverOptions().setSeed(42).setMaxIterations(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 (`LineOptExamples.loadBalancing`) declares the routing split as the decision variable:

```
OptimizationProblem p = new OptimizationProblem(model);
List<Node> targets = new ArrayList<Node>();
targets.add(fast);
targets.add(slow);
p.addVariable(new RoutingProbabilities(jobs, source, targets));
p.setObjective(new MinimizeSystemResponseTime(jobs));

OptimizationResult r = p.solve(
        new LineOptSolverOptions().setSeed(42).setMaxIterations(60));
double[] probs = (double[]) r.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 (`LineOptExamples.populationSizing`):

```
Network model = new Network("Interactive");
Delay think = new Delay(model, "Think");
Queue server = new Queue(model, "AppServer", SchedStrategy.PS);
ClosedClass users = new ClosedClass(model, "Users", 1, think);
think.setService(users, new Exp(1.0));
server.setService(users, new Exp(2.0));
model.link(Network.serialRouting(think, server));

OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new JobPopulation(users, 1, 50));
p.setObjective(new MinimizeCost(null, null, null, null));
p.addConstraint(new ResponseTimeConstraint(server, users, 2.0));

OptimizationResult r = new BisectionSolver(p, "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 `addScenario`; the solver enforces every constraint on all scenarios (`LineOptExamples.robustSizing`):

```
Network base = mmc(3.0);   // average load
Network peak = mmc(4.5);   // peak load scenario
Queue queue = (Queue) base.getNodes().get(1);

OptimizationProblem p = new OptimizationProblem(base);
p.addVariable(new ServerAllocation(queue, 1, 12));
p.setObjective(new MinimizeCost(serverCost, null, null, null));
p.addConstraint(new UtilizationConstraint(queue, 0.5));
p.addScenario(peak, 1.0);

OptimizationResult r = new BisectionSolver(p).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 (`LineOptExamples.paretoFrontier`):

```
double[] eps = {0.3, 0.4, 0.5, 0.6, 0.75, 0.9};
ParetoSweep sweep = new ParetoSweep(p, new ParetoSweep.ConstraintFactory() {
    public Constraint make(double epsilon) {
        return new UtilizationConstraint(queue, epsilon);
    }
}, eps, "bisection");
sweep.solve();
for (ParetoPoint pt : sweep.getFrontier()) {
    System.out.println("  util <= " + pt.epsilon + " -> " + pt.objectiveValue
            + " (" + pt.result.getVariableValue("Server_servers") + ")");
}
```

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. Frontier points are returned as a list of `ParetoPoint` objects carrying the bound (`epsilon`), the optimal objective (`objectiveValue`) and the underlying `OptimizationResult` (`result`); plotting is left to the caller.

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 (`LineOptExamples.decomposition`):

```
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServerAllocation(queue, 1, 10));
p.addVariable(new ServiceRate(queue, jobs, 1.0, 4.0));
p.setObjective(new MinimizeCost(serverCost, rateCost, null, null));
p.addConstraint(new ResponseTimeConstraint(queue, jobs, 0.5));

DecompositionWorkflow workflow = p.decompose();
workflow.autoDecompose();
workflow.setSolverOptions(new LineOptSolverOptions().setSeed(42));
WorkflowResult r = workflow.solveSequential(4, 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 blockwise 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 `jline.examples.opt.LineOptExamples`: 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.

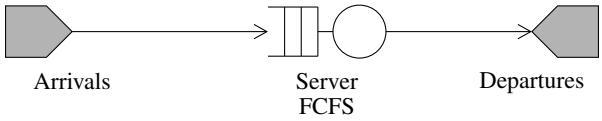

Figure 9.1: Topology of `opt_bisection_sizing`: an open network over 1 queueing station.

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.

```
Network model = mmc(3.0);
Queue queue = (Queue) model.getNodes().get(1);
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServerAllocation(queue, 1, 10));
p.setObjective(new MinimizeCost(cost(queue, 10.0), null, null, null));
p.addConstraint(new UtilizationConstraint(queue, 0.5));
OptimizationResult r = new BisectionSolver(p).solve();
System.out.println("[bisection_sizing] servers=" + r.getVariableValue("Server_servers")
        + " cost=" + r.objectiveValue + " probes=" + r.modelEvaluations);
```

Running this edition prints

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[bisection_sizing] servers=6 cost=60.0 probes=4
```

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.

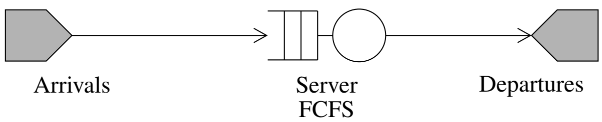

Figure 9.2: Topology of `opt_decomposition`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Network model = mmc(3.0);
Queue queue = (Queue) model.getNodes().get(1);
OpenClass jobs = (OpenClass) model.getClasses().get(0);
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServerAllocation(queue, 1, 10));
p.addVariable(new ServiceRate(queue, jobs, 1.0, 4.0));
p.setObjective(new MinimizeCost(cost(queue, 10.0), cost(queue, 20.0), null, null));
p.addConstraint(new ResponseTimeConstraint(queue, jobs, 0.5));
DecompositionWorkflow workflow = p.decompose();
```

```
workflow.autoDecompose();
workflow.setSolverOptions(new LineOptSolverOptions().setSeed(42));
WorkflowResult r = workflow.solveSequential(4, 1e-3);
System.out.println("[decomposition] converged=" + r.converged
        + " servers=" + r.getFinalVariableValue("Server_servers")
        + " rate=" + r.getFinalVariableValue("Server_Jobs_rate")
        + " obj=" + r.finalObjective);
```

Running this edition prints

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
WARNING: [getAvg] The model has unstable queues (utilization >= 1); station utilization is reported capped
      at 1.0, queue length and response time as Inf.
Successful method execution completed by SolverMVA
WARNING: [getAvg] Message casted more than once, repetitions will not be printed on screen for 60 seconds.

... (1284 captured-output lines omitted; rerun the source example for the full output)

Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[decomposition] converged=true servers=7 rate=2.000519245058782 obj=110.01038490117564
```

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.

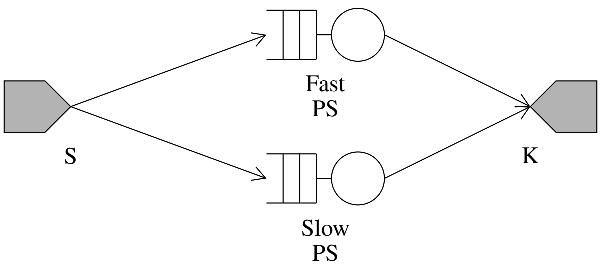

Figure 9.3: Topology of `opt_load_balancing`: an open network over 2 queueing stations.

The example is built and solved as follows.

```
Network model = new Network("LoadBalance");
Source source = new Source(model, "S");
Queue fast = new Queue(model, "Fast", SchedStrategy.PS);
Queue slow = new Queue(model, "Slow", SchedStrategy.PS);
Sink sink = new Sink(model, "K");
OpenClass jobs = new OpenClass(model, "Jobs");
source.setArrival(jobs, new Exp(2.0));
fast.setService(jobs, new Exp(4.0));
```

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`.

```
slow.setService(jobs, new Exp(2.5));
model.addLink(source, fast);
model.addLink(source, slow);
model.addLink(fast, sink);
model.addLink(slow, sink);
OptimizationProblem p = new OptimizationProblem(model);
List<Node> targets = new ArrayList<Node>();
targets.add(fast);
targets.add(slow);
p.addVariable(new RoutingProbabilities(jobs, source, targets));
p.setObjective(new MinimizeSystemResponseTime(jobs));
OptimizationResult r = p.solve(new LineOptSolverOptions().setSeed(42).setMaxIterations(60));
double[] probs = (double[]) r.getVariableValue("Jobs_routing_from_S");
System.out.println("[load_balancing] fast=" + probs[0] + " slow=" + probs[1]);
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 42.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 33.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 46.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 46.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 63.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 85.

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

Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 34.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 41.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[load_balancing] fast=0.7362332547246861 slow=0.26376674527531385
```

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.

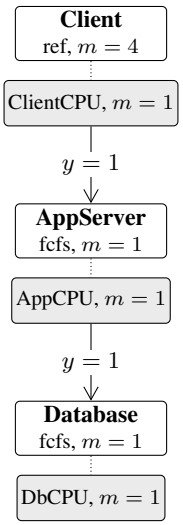

Figure 9.4: Call graph of `opt_lqn_host_demand`: 3 tasks over 3 processors, drawn with the reference task on top and a called task below its caller; the shaded box under each task is the processor it runs on.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

9.9.5 `opt_pareto_frontier`

The cost-performance tradeoff frontier, swept rather than solved at one weighting, through `ParetoSweep`.

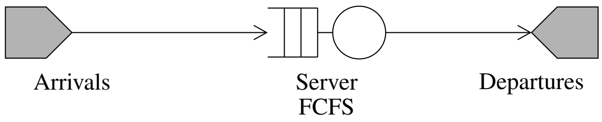

Figure 9.5: Topology of `opt_pareto_frontier`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Network model = mmc(3.0);
final Queue queue = (Queue) model.getNodes().get(1);
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServerAllocation(queue, 1, 10));
p.setObjective(new MinimizeCost(cost(queue, 10.0), null, null, null));
double[] eps = {0.3, 0.4, 0.5, 0.6, 0.75, 0.9};
ParetoSweep sweep = new ParetoSweep(p, new ParetoSweep.ConstraintFactory() {
    public Constraint make(double epsilon) {
        return new UtilizationConstraint(queue, epsilon);
    }
}, eps, "bisection");
sweep.solve();
System.out.println("[pareto_frontier]");
for (ParetoPoint pt : sweep.getFrontier()) {
    System.out.println("  util <= " + pt.epsilon + " -> " + pt.objectiveValue
            + " (" + pt.result.getVariableValue("Server_servers") + ")");
}
```

Running this edition prints

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
```

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.

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

Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[pareto_frontier]
  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 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.

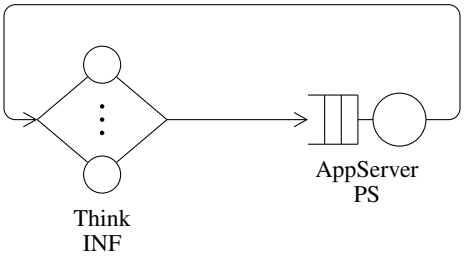

Figure 9.6: Topology of `opt_population_sizing`: a closed network over 2 queueing stations.

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.

```
Network model = new Network("Interactive");
Delay think = new Delay(model, "Think");
Queue server = new Queue(model, "AppServer", SchedStrategy.PS);
ClosedClass users = new ClosedClass(model, "Users", 1, think);
think.setService(users, new Exp(1.0));
server.setService(users, new Exp(2.0));
model.link(Network.serialRouting(think, server));
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new JobPopulation(users, 1, 50));
p.setObjective(new MinimizeCost(null, null, null, null));
p.addConstraint(new ResponseTimeConstraint(server, users, 2.0));
OptimizationResult r = new BisectionSolver(p, "max_feasible").solve();
System.out.println("[population_sizing] users=" + r.getVariableValue("Users_population")
        + " evals=" + r.modelEvaluations);
```

Running this edition prints

```
MVA analysis [method: default/egflin; type: approximate, deterministic; lang: java; env: python] completed
      in <t>s. Iterations: 21.
Successful method execution completed by SolverMVA
```

```
Successful method execution completed by SolverMVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: exact; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/exact; type: exact, deterministic; lang: java; env: python] completed in <t>
    s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[population_sizing] users=5 evals=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.

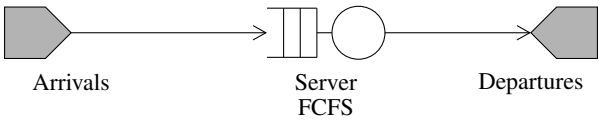

Figure 9.7: Topology of `opt_robust_sizing`: an open network over 1 queueing station.

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`.

The example is built and solved as follows.

```
Network base = mmc(3.0);
Network peak = mmc(4.5);
Queue queue = (Queue) base.getNodes().get(1);
OptimizationProblem p = new OptimizationProblem(base);
p.addVariable(new ServerAllocation(queue, 1, 12));
p.setObjective(new MinimizeCost(cost(queue, 10.0), null, null, null));
p.addConstraint(new UtilizationConstraint(queue, 0.5));
p.addScenario(peak, 1.0);
OptimizationResult r = new BisectionSolver(p).solve();
System.out.println("[robust_sizing] servers=" + r.getVariableValue("Server_servers")
        + " cost=" + r.objectiveValue);
```

Running this edition prints

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
```

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[robust_sizing] servers=9 cost=90.0
```

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.

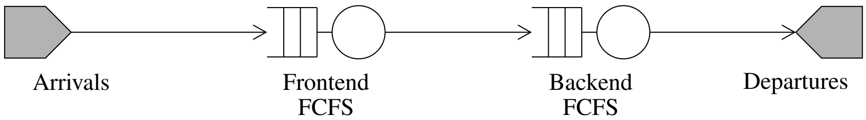

Figure 9.8: Topology of `opt_sensitivity_report`: an open network over 2 queueing stations.

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. This reference example has no source implementation in this edition and is not runnable here. Consult the corresponding language directory in the LINE repository for the current porting status. No execution output has been captured for this language edition. Run the identified implementation to obtain its results.

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.

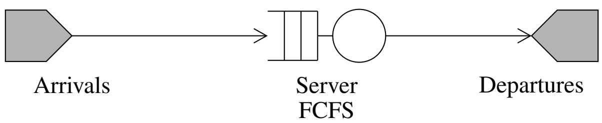

Figure 9.9: Topology of `opt_server_sizing`: an open network over 1 queueing station.

The example is built and solved as follows.

```
Network model = mmc(3.0);
Queue queue = (Queue) model.getNodes().get(1);
```

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`.

```java
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServerAllocation(queue, 1, 10));
p.setObjective(new MinimizeCost(cost(queue, 10.0), null, null, null));
p.addConstraint(new UtilizationConstraint(queue, 0.5));
OptimizationResult r = p.solve(new LineOptSolverOptions().setSeed(42));
System.out.println("[server_sizing] servers=" + r.getVariableValue("Server_servers")
        + " cost=" + r.objectiveValue + " evals=" + r.modelEvaluations);
```

Running this edition prints

```
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
WARNING: [getAvg] The model has unstable queues (utilization >= 1); station utilization is reported capped
      at 1.0, queue length and response time as Inf.
Successful method execution completed by SolverMVA
WARNING: [getAvg] Message casted more than once, repetitions will not be printed on screen for 60 seconds.

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

Successful method execution completed by SolverMVA
MVA analysis [method: default/mmk; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[server_sizing] servers=6 cost=60.0 evals=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.

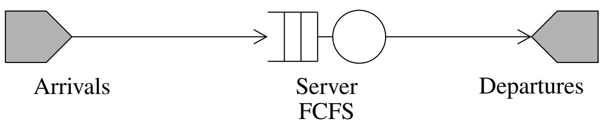

Figure 9.10: Topology of `opt_service_rate`: an open network over 1 queueing station.

The example is built and solved as follows.

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`.

```
Network model = new Network("MM1");
Source source = new Source(model, "Arrivals");
Queue queue = new Queue(model, "Server", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Departures");
OpenClass jobs = new OpenClass(model, "Jobs");
source.setArrival(jobs, new Exp(3.0));
queue.setService(jobs, new Exp(4.0));
model.link(Network.serialRouting(source, queue, sink));
OptimizationProblem p = new OptimizationProblem(model);
p.addVariable(new ServiceRate(queue, jobs, 3.5, 8.0));
p.setObjective(new MinimizeCost(null, cost(queue, 20.0), null, null));
p.addConstraint(new ResponseTimeConstraint(queue, jobs, 0.5));
OptimizationResult r = p.solve(new LineOptSolverOptions().setSeed(42).setMaxIterations(60));
System.out.println("[service_rate] rate=" + r.getVariableValue("Server_Jobs_rate")
        + " cost=" + r.objectiveValue);
```

Running this edition prints

```
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.

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

Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
MVA analysis [method: default/mm1; type: exact, deterministic; lang: java; env: python] completed in <t>s.
Successful method execution completed by SolverMVA
Successful method execution completed by SolverMVA
[service_rate] rate=5.000778867588173 cost=100.01557735176347
```

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.