LINE: Queueing Analysis Algorithms

A Primer: R Edition

Last revision: November 28, 2026
QORE Lab, Imperial College London

Contents

1 A guided tour of LINE **1**

1.1 Overview and installation . . . 1
1.1.1 Obtaining and installing LINE . . . 1
1.1.2 Software requirements . . . 2
1.1.3 Checking the installation . . . 2
1.2 A first model . . . 3
1.2.1 The M/M/1 queue . . . 3
1.2.2 Reading the results . . . 3
1.3 Network models . . . 4
1.3.1 Nodes, scheduling and classes . . . 4
1.3.2 Distributions . . . 4
1.3.3 Open and closed networks . . . 4
1.3.4 Routing and advanced features . . . 5
1.4 Analysis . . . 5
1.4.1 Performance metrics . . . 5
1.4.2 Solvers . . . 6
1.5 Layered queueing networks . . . 7
1.6 Random environments . . . 8
1.7 Further resources . . . 8

2 Tutorials **10**

2.1 Tutorial 1: A M/M/1 queue . . . 10
2.2 Tutorial 2: A multiclass M/G/1 queue . . . 11
2.3 Tutorial 3: Machine interference problem . . . 13
2.4 Tutorial 4: Round-robin load-balancing . . . 14
2.5 Tutorial 5: Modelling a reentrant line . . . 16
2.6 Tutorial 6: A queueing network with caching . . . 17
2.7 Tutorial 7: Delayed-hit caches with a retrieval system . . . 20
2.8 Tutorial 8: Response time distribution and percentiles . . . 21
2.9 Tutorial 9: Optimizing a performance metric . . . 23
2.10 Tutorial 10: Studying a departure process . . . 24
2.11 Tutorial 11: Basic layered queueing network . . . 25
2.12 Tutorial 12: Random environments . . . 26
2.13 Tutorial 13: Posterior analysis . . . 27
2.14 Tutorial 14: Open cluster . . . 29
2.15 Tutorial 15: A loss network . . . 30
2.16 Tutorial 16: A queueing Petri net . . . 31
2.17 Tutorial 17: A colored generalized stochastic Petri net . . . 33

Chapter 1

A guided tour of LINE

1.1 Overview and installation

LINE is an open-source software package for the analysis of queueing models by means of analytical methods and simulation. It is developed by the QORE lab at Imperial College London and distributed under the BSD-3 license. The package solves single queueing systems (M/M/1, M/M/k, M/G/1, ...), open and closed queueing networks, layered queueing networks, and models embedded in random environments. Models are solved either by LINE's native algorithms or through external solvers such as JMT and LQNS.

`LDES`, the discrete-event simulation engine of LINE, is built to span the entire feature space of the package, so that every approximation LINE publishes can be validated against a sample path of the same model, expressed once. Several analytical solvers are included alongside it to accelerate the evaluation of specific classes of models, and their solutions can in turn warm-start an `LDES` run, placing it near steady state already at the start of the simulation (Sec. 1.4.2).

In this primer, we give a guided tour of the package. We first build and solve a single queue, so as to introduce the modeling workflow and the format of the results. We then generalize the workflow to arbitrary network models and survey the available solvers. Lastly, we introduce layered queueing networks and models embedded in random environments. Advanced topics, such as transient and distribution analysis, fall beyond the scope of this primer and are treated in the user manual.

The LINE documentation is organized into five complementary documents for each language edition: (1) this *Primer*, which introduces the modeling workflow and walks through 17 self-contained tutorials; (2) the *User Manual* (`LINE-user-*.pdf`), the complete reference covering all formalisms, distributions, scheduling disciplines, solvers, parameter inference, and optimization; (3) the *Examples Manual* (`LINE-examples-*.pdf`), a runnable catalog of every script in the `examples/` gallery; (4) the *Internals Manual* (`LINE-internals-*.pdf`), which documents the mathematical algorithms and internal architecture of each solver; and (5) the one-page *Cheatsheet* (`LINE-cheatsheet-*.pdf`) for quick syntax lookup.

1.1.1 Obtaining and installing LINE

LINE's R edition is the package `linesolver`. It is NOT on CRAN, because the engine behind it is one translation unit that peaks at 17.9 GB of resident memory and some twenty-seven minutes of CPU, against a check budget measured in minutes. Download the R archive `line-<version>-r.zip` of the current release from the R card on https://line-solver.sourceforge.net, or from the release folder at https://sourceforge.net/projects/line-solver/files/, and unzip it. The source tarball `r/linesolver_<version>.0.tar.gz` it carries installs from a prebuilt engine or from source:

```
# preferred: point the install at an engine that already exists
LINE_CAPI_LIB=/opt/line/lib/libline_capi.a \
```

```
LINE_CAPI_INCLUDE=/opt/line/include \
    R CMD INSTALL r/linesolver_*.tar.gz

# or decline the engine and install the document half only
LINE_CAPI_BUILD=no R CMD INSTALL r/linesolver_*.tar.gz
```

The second form is a supported configuration, not a degraded one: it builds models and reads and writes the portable `line-model` document, which makes R a first-class *producer* of models the other four editions solve on a machine with no LINE build at all. The package says which half it has at attach, and `capi_available()` reports it.
The latest releases and the source code are available from http://line-solver.sf.net.

1.1.2 Software requirements

Three external tools are optional. Each one backs a specific part of LINE and none of them is needed by the native solvers:

- *Java Modelling Tools* (JMT, http://jmt.sf.net), version 1.3.0 or later, is required by the JMT simulation solver.
- *LQNS* (https://github.com/layeredqueuing/V6), version 6.2.28 or later, is required by the LQNS wrapper, for layered models and for the `qnsolver` methods it offers on flat networks. The `lqns`, `lqsim` and `qnsolver` commands must be on the system path.
- The *symbolic backend*, a SageMath computer algebra system behind a small REST service, is used by the symbolic methods of the CTMC and fluid solvers. It requires Docker and is fetched once with `docker pull imperialqore/line-sage-rest:latest`, after which LINE starts and stops the service itself.

The user manual gives the full list, with the version constraints and the platform-specific installation commands.

1.1.3 Checking the installation

Every distribution ships an environment check that reports which optional dependencies (a Java runtime, the LQNS binaries, JMT, the C++ `line-cli` engine, and the SageMath symbolic backend) are reachable. It reports only: a missing JMT is named, not downloaded, so that running the check does not acquire the dependency it is reporting on. Run it once after installing:

```
line_version()
```

```
$package   [1] "3.0.9.0"
$available [1] TRUE
$abi       [1] 1
$cli       [1] "0.1.0"
$features  $lapack TRUE  $highs FALSE  $gmp FALSE  $symengine FALSE
```

`features` is the useful half: it says what this build will refuse BEFORE you run something it refuses. The external-program solvers (`LDES`, `JMT`, `LN` with `lqns`, `LQNS`) are not reported there, because they are found at call time and each refuses by naming the program it could not run.
A missing dependency is reported as a warning rather than an error, because it disables a subset of the solvers and leaves the rest of LINE usable. External services and engines are otherwise resolved by the method that requests them: symbolic methods connect to the configured SageMath backend, the JMT wrapper resolves a Java runtime and `JMT.jar`, and the LQNS wrapper requires the corresponding commands on `PATH`.

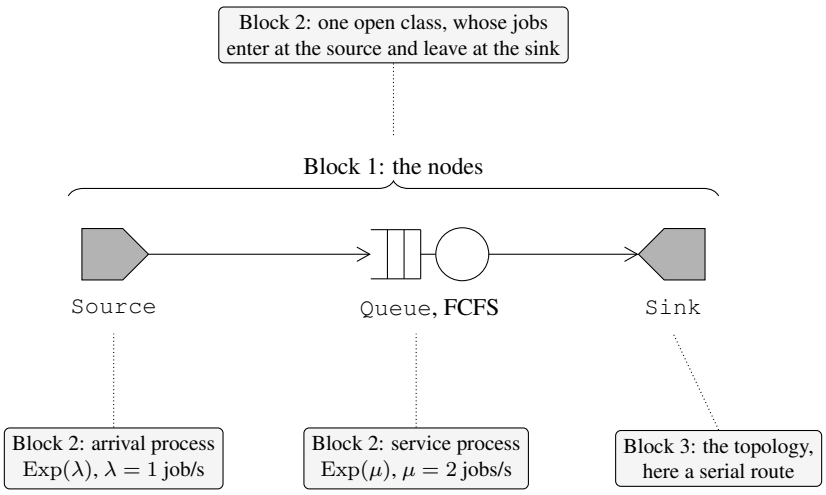


Figure 1.1: The M/M/1 queue, annotated with the blocks every LINE model is built from. A Poisson stream of rate λ feeds one FCFS station with exponential service of rate μ, and completed jobs leave through the sink. The fourth block, the solution, is the choice of solver and has no counterpart in the diagram.

1.2 A first model

Every LINE model is assembled from the same four conceptual blocks: (1) the *nodes* of the network, among them the *stations* where jobs queue and are served; (2) the *job classes*, together with the *arrival* and *service* processes attached to each station and class; (3) the *topology*, that is, the routes the jobs follow; and (4) the *solution*, in which the finished model is handed to a *solver* that returns performance metrics. The tutorial listings develop those blocks in reading order; the directly runnable companion files retain ordinary program structure and do not all repeat the four labels as comments. This chapter presents the concepts and the vocabulary that go with them; the scripts that realize them in the language of this edition are collected in Chapter 2, one worked model per section.

1.2.1 The M/M/1 queue

The smallest model worth building is the M/M/1 queue: a single first-come first-served station, fed by a Poisson arrival stream, whose service times are exponentially distributed. Figure 1.1 shows it with the first three blocks named on the diagram. The nodes are a `Source`, a `Queue` and a `Sink`; a single open class of jobs enters at the source and leaves at the sink, arriving at rate $\lambda = 1$ and being served at rate $\mu = 2$; and the topology is the serial route that connects the three nodes in order. The fourth block, the solution, does not appear in the diagram: it is the choice of solver.
With these parameters elementary queueing theory gives a utilization of $\rho = \lambda/\mu = 0.5$ and a mean response time of $1/(\mu - \lambda) = 1$s. An analytical LINE solver returns those values exactly; the LDES simulation in Tutorial 1 estimates them within sampling error. The script that builds and solves this model is Tutorial 1 (Section 2.1).

1.2.2 Reading the results

Every average-metric solver returns a table with one row per (station, class) pair. The main columns are as follows:

Column	Meaning
QLen	mean number of jobs at the station (queue length)
Util	fraction of the servers kept busy; at a delay (infinite-server station), the mean number in service
RespT	mean response time per visit
ResidT	mean residence time (response time weighted by visits)
ArvR	mean arrival rate
Tput	mean throughput

A row is retrieved by the station and class *objects* that produced it, or by their names, rather than by position, so that a script never depends on the order in which the nodes were declared; filtering by a station alone, or by a class alone, returns every row that mentions it. The selection idioms of this edition are shown in Section 2.1.
Stochastic solvers additionally accept a `seed`, which makes a run reproducible, and an `events` (or `samples`) budget, which trades accuracy against time. Verbosity is controlled globally through .

1.3 Network models

We now generalize the workflow of Section 1.2 to arbitrary networks. The `Network` object is the container for all model elements, which register themselves with the model passed to their constructor. The four-block structure carries over unchanged; only the number of nodes, classes, and routes grows.

1.3.1 Nodes, scheduling and classes

Node	Role
Source / Sink	external arrival / departure (open classes)
Queue	queueing station with a scheduling discipline
Delay	infinite-server (pure think time / delay)
Router	pure routing / load-balancing node
ClassSwitch	changes the class of transiting jobs
Cache	caching node (hit/miss with a replacement policy)
Fork / Join	fork-join parallelism
Place / Transition	stochastic Petri-net elements

The scheduling discipline is passed to the `Queue` constructor as a `SchedStrategy` value: `FCFS` (first-come first-served), `PS` (processor sharing), `INF` (infinite server), `LCFS`, `SIRO` (random order), `HOL` (head-of-line priority), `SEPT`/`LEPT`, and `DPS`/`GPS`. A queue's server count is set with `setNumberOfServers`. Two class kinds exist: `OpenClass` jobs enter at a `Source` and leave at a `Sink`; `ClosedClass` jobs circulate among a fixed population with a designated reference station. Open and closed classes may coexist in a mixed model.

1.3.2 Distributions

Arrival and service processes are distribution objects: `Exp` (exponential), `Erlang`, `HyperExp`, `Cox2`/`APH` (phase-type), `Det` (deterministic), `Uniform`, `Gamma`, `Pareto`, `Lognormal`, `MAP`/`MMPP2` (Markovian arrivals), and `Replayer` (empirical trace). Nonexponential timing is handled by phase-type expansion or by fitting: matches a target mean and squared coefficient of variation.

1.3.3 Open and closed networks

A network is *open* when jobs enter it from a `Source` and leave it at a `Sink`, so that the population present at any instant is itself a random variable; it is *closed* when a fixed population circulates indefinitely, and *mixed* when open and closed classes coexist. A closed class names a *reference station*, which marks the point at which a job is considered to have completed one cycle through the network; system response time and system throughput are measured against it.

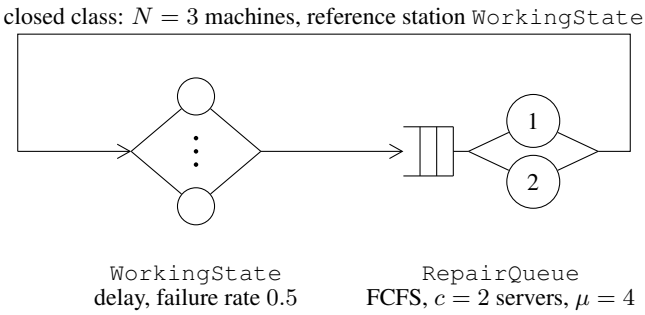


Figure 1.2: A closed network: the machine-repairman model. A fixed population of $N = 3$ machines alternates between a working state, modelled as a delay, and a two-server FCFS repair station. No job enters or leaves, so exactly N jobs are present at all times.

Figure 1.2 shows a closed network of the kind that recurs throughout queueing theory, the machine-repairman model: three machines alternate between a working state, modelled as a delay because a machine never waits to start working, and a repair station staffed by two repairmen, which is an ordinary FCFS queue with two servers. Machines fail at rate 0.5 and are repaired at rate 4. Because the state space is small, the model can be solved exactly by generating its Markov chain: on average 2.66 of the three machines are working and 0.34 are under repair, at a repair-station utilization of 0.167 per server. The corresponding script is Tutorial 3 (Section 2.3).

1.3.4 Routing and advanced features

Serial chains are declared with a serial-routing shorthand; point-to-point links are added one at a time, and per-node behaviour is set with a `RoutingStrategy` such as `RAND`, `RROBIN` (round robin), `PROB` (probabilistic), or `JSQ` (join shortest queue). Class-dependent routing uses a routing matrix obtained from the model and populated per class. LINE also supports finite-capacity regions, reward models, caches with replacement policies, class switching, and stochastic Petri nets; their treatment falls beyond the scope of this primer and is given in the user manual.

1.4 Analysis

1.4.1 Performance metrics

The primary analysis is steady-state average performance. The metrics of Section 1.2 are reported per station and class; Figure 1.3 names them on a network, so that each can be read off the place in the model it refers to.

Response times, throughputs and queue lengths are also reported end-to-end. These are the *system* metrics `SysRespT`, `SysTput` and `SysQLen`, indexed by chain rather than by station-class pair: a system response time is the time a job spends in the network, measured from its arrival at the `Source` for an open chain or from its departure from the reference station for a closed one, until it reaches the `Sink` or returns to that station.

Beyond averages, LINE reports transient averages, response-time distributions and, for Markovian models, the underlying state space and generator, from which state probabilities are read; these fall beyond the scope of this primer and are covered in the user manual.

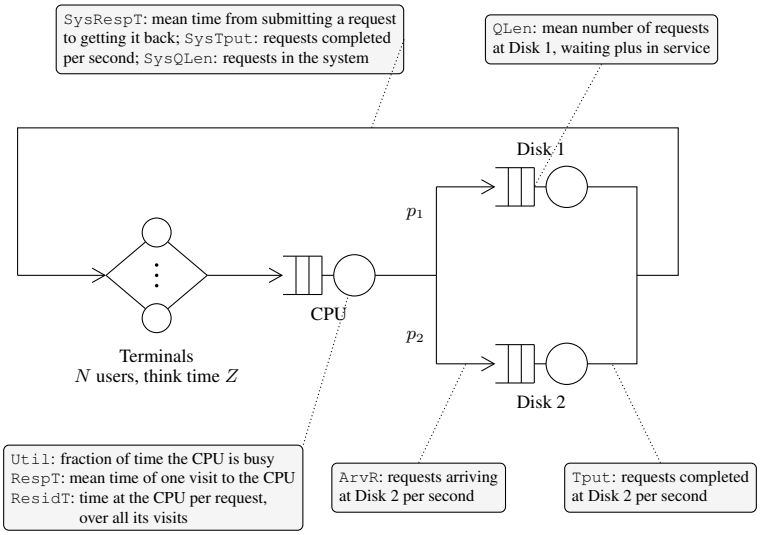


Figure 1.3: The metrics LINE reports, named on a closed network in which N users at terminals submit requests that are served by a CPU and by one of two disks before returning to their user. `QLen`, `Util`, `RespT`, `ResidT`, `ArvR` and `Tput` are reported per station and class; the system metrics are reported per chain.

1.4.2 Solvers

A solver takes a model and a set of options and returns metrics. All solvers are invoked uniformly, as *Solver(model, options)* followed by the method that selects the metrics wanted, and since every solver consumes the same model object unchanged, running two solvers and cross-validating their results is good practice. Solvers fall into three groups. *Native solvers* implement an analysis algorithm inside LINE. *Metasolvers* do not analyze a model themselves: they decompose or replicate it and delegate each piece to another solver, which the caller supplies. *Wrappers* translate the model into the input format of an external tool, run it as a separate process, and read its results back.

Solver	Method	Best for
Native solvers		
`AG`	agent-based decomposition	product-form models with synchronization
`BA`	bound analysis	guaranteed bounds, optimization loops
`CTMC`	continuous-time Markov chain	exact, small state spaces
`FLD`	fluid / mean-field ODE	large populations, transient
`LDES`	discrete-event simulation (native)	non-Markovian timing, widest features
`MAM`	matrix-analytic methods	MAP/PH open models
`MVA`	mean value analysis	product-form open/closed, fast
`NC`	normalizing constant	large closed populations
`SSA`	stochastic simulation (native)	general, any feature
Metasolvers		
`AUTO`	automatic solver selection	when no solver is chosen by hand
`ENV`	random-environment blending	environment models (Sec. 1.6)
`LN`	layer decomposition	layered models (Sec. 1.5)
`UQ`	prior-weighted expansion	models with uncertain parameters
External solvers (wrappers)		
`JMT`	discrete-event simulation (JMT)	validation, general
`LQNS`	`lqns` / `lqsim` / `qnsolver`	layered models; product-form multiserver (`qns.*`)

A metasolver is given the inner solver to use as an argument, so the same layered or environment model can be evaluated at very different costs by changing that one argument. Wrappers are available only when their engines can be resolved at run time: `JMT` requires a Java runtime and `JMT.jar`, while

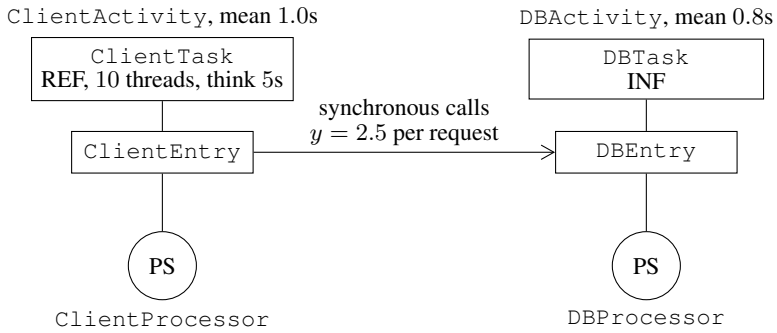


Figure 1.4: A layered queueing network: a two-tier client–server application. Each task offers one entry, runs one activity, and is hosted by its own processor; the client activity issues on average 2.5 synchronous calls per request to the database entry.

LQNS requires the separately distributed `lqns`, `lqsim` and `qnsolver` commands. Their absence does not affect the native solvers. Solver support is also model-dependent: "general" in this table is a selection guideline, not a guarantee that every feature is implemented by that solver in every language edition.

As a rule of thumb, we recommend CTMC for small exact models; MVA/NC for product-form networks at scale; BA when a guaranteed interval is worth more than a point estimate; FLD for very large populations or transients; MAM for Markovian-arrival or phase-type open models; and SSA, LDES or JMT when a feature is not supported analytically, with LDES carrying the widest feature envelope of the three.

The two groups also combine. The solution of an analytical solver can be used to *warm-start* a simulation: the initial state of an LDES or SSA run is seeded with a job placement taken from a cheaper solver, so that the run starts near its steady state and no samples are spent traversing the initial transient. An exact auxiliary solver contributes the mode of its stationary distribution, an approximate one its rounded mean queue lengths, with the closed-class populations conserved. The placement is set with the `ldes_initsol` solver option, and the *Warm start* section of the user manual documents which solver may seed which, and what the mechanism costs and buys.

1.5 Layered queueing networks

A `LayeredNetwork` (LQN) captures software contention on top of hardware contention. Its elements are *processors* (hardware servers), *tasks* (software resources with a multiplicity), *entries* (the service interfaces a task offers), and *activities* (units of work that issue synchronous or asynchronous calls to other entries). A synchronous call blocks the caller until the callee replies, so a task can be held by a resource it does not itself contend for – which is the phenomenon a flat queueing network cannot express.

Figure 1.4 shows a two-tier client–server application. The client task has ten threads, each thinking for 5s, doing 1s of its own work per request, and issuing on average 2.5 synchronous calls to the database entry; the database serves each call in 0.8s. Each task runs on its own processor-sharing processor.

The native LN solver decomposes the model into layers and applies a supplied per-layer solver, iterating until the layers agree; the external LQNS solver may be used instead when available. Results are reported per element rather than per station and class, one row for each processor, task, entry and activity, tagged with its type. A metric that is undefined for an element type is reported as NaN: processors carry no queue length or response time of their own, tasks report a residence time rather than a per-visit response time, and entries report the converse.

Solving this model with LN over MVA shows the database processor saturated at a utilization of 0.998, which is what caps the client throughput at 0.499 requests per second, against a client-side response time of 15.0s per request. The script and its full result table are Tutorial 11 (Section 2.11).

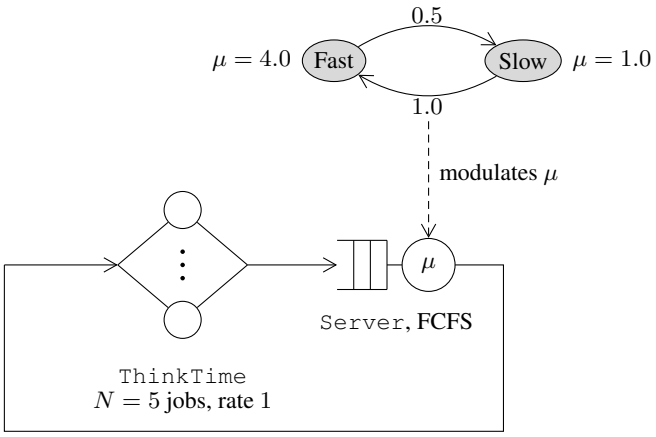


Figure 1.5: A model in a random environment: a two-state Markov chain alternates between a fast and a slow mode, each a complete stage network, and modulates the service rate of the queue. The environment is fast $2/3$ of the time and slow $1/3$ of the time.

1.6 Random environments

A random environment models a system whose parameters change according to an underlying stochastic process. Each *stage* is a complete network with its own parameters, and stage transitions follow specified distributions. The `ENV` metasolver takes a factory producing an inner solver for each stage and returns metrics averaged over the stationary distribution of the environment.

Figure 1.5 shows the pattern at its simplest: a two-station closed network with five jobs, whose server alternates between a fast mode ($\mu = 4$) and a slow mode ($\mu = 1$). Only the service rate differs between the two stages; everything else is the same network. The environment leaves the fast stage at rate 0.5 and the slow stage at rate 1.0, so it spends $2/3$ of the time fast and $1/3$ slow, and the reported metrics are averages over that stationary distribution rather than over either stage alone. The script is Tutorial 12 (Section 2.12).

1.7 Further resources

The `line-cli.py` script solves models without writing code, reading LINE's native JSON format as well as JMT's JSIMG and LQNS's LQNX, and printing tables, JSON, or CSV:

```
python line-cli.py solve example.json --solver mva
```

Here `example.json` stands for any model in LINE's JSON format; a model built in a script is written to one by . Every metric is reported on its own row, station by station and class by class, followed by the system metrics per chain. The same script also lists what is installed, each solver with the model formats it can read:

```
python line-cli.py list solvers
```

LINE is also available as a Model Context Protocol (MCP) server, letting LLM tools such as Claude Code build and solve models through natural language (`pip install line-solver`). The complete references are the LINE user manuals (`LINE-user-matlab.pdf`, `LINE-user-java.pdf`, `LINE-user-python.pdf` and `LINE-user-cpp.pdf`) with their Sphinx (MATLAB and Python), Javadoc (Java) and Doxygen (C++) API references, plus the MCP getting-started guide, all linked from http://line-solver.sf.net.

Chapter 2 works through seventeen complete models in the language of this edition, beginning with

the M/M/1 queue of Section 1.2. For many more worked examples, covering additional queueing systems, layered and mixed networks, transient and distribution analysis, inference, and optimization, we refer the reader to the LINE user manual for this language (), which expands every topic in this primer in greater depth, and to the LINE example manual (), which catalogues the runnable example suite.

Chapter 2

Tutorials

This chapter walks through seventeen complete models, from a single M/M/1 queue to layered, cached, and Petri-net models. Each tutorial states the model, develops the code in reading order, and comments the resulting metrics. The Python source distribution also contains directly runnable versions in `python/examples/gettingstarted`; installed wheels expose the same files as `line_solver.examples.gettingstarted` modules. Those files include the imports and resource handling needed to run them from any working directory. The modelling constructs used here are documented in full in the LINE user manual (`LINE-user-matlab.pdf`, `LINE-user-java.pdf`, `LINE-user-python.pdf`, `LINE-user-cpp.pdf`), and the solution algorithms in the language-specific LINE internals manual ().

2.1 Tutorial 1: A M/M/1 queue

The M/M/1 queue is a classic model of a queueing system where jobs arrive into an infinite-capacity buffer, wait to be processed in first-come-first-served (FCFS) order, and then leave after service completion. Arrival and service times are assumed to be independent and exponentially distributed random variables.

In this example, we wish to compute average performance measures for the M/M/1 queue. We assume that arrivals come in at rate $\lambda = 1$ job/s, while service has rate $\mu = 2$ job/s. It is known from theory that the exact value of the server utilization in this case is $\rho = \lambda/\mu = 0.5$, i.e., 50%, while the mean response time for a visit is $R = 1/(\mu - \lambda) = 1$s. We wish to verify these values using LDES-based discrete-event simulation, instantiated through LINE.

Laid out in the four blocks of Section 1.2, which every script in this chapter marks in its comments, the M/M/1 model is solved as follows

```
model <- Network$new('M/M/1')
# Block 1: nodes
source <- Source$new(model, 'mySource')
queue  <- Queue$new(model, 'myQueue', SchedStrategy$FCFS)
sink   <- Sink$new(model, 'mySink')
# Block 2: classes
oclass <- OpenClass$new(model, 'myClass')
source$set_arrival(oclass, Exp(1))
queue$set_service(oclass, Exp(2))
# Block 3: topology
model$link(serial_routing(source, queue, sink))
# Block 4: solution
avg_table <- LDES$new(model, d = 23000, samples = 10000)$avg_table()
print(avg_table)
# one cell, by column, station and class
tget(avg_table, 'QLen', 'myQueue', 'myClass')
```

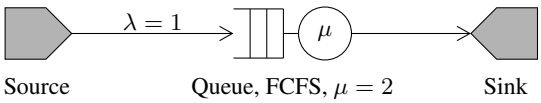


Figure 2.1: The M/M/1 queue of Tutorial 1: a Poisson source feeds a single FCFS station with exponential service, and completed jobs leave through the sink.

In the example, `source` and `sink` are arrival and departure points of jobs; `queue` is a queueing station with FCFS scheduling; `jobclass` defines an open class of jobs that arrive, get served, and leave the system; `Exp(2.0)` defines an exponential distribution with service rate $\mu = 2.0$; finally, the call solves for average performance measures with LINE's own discrete-event simulator, LDES, using for reproducibility a specific seed for the random number generator.

The result is a table with mean performance measures including: the number of jobs in the station either queueing or receiving service (`QLen`); the utilization of the servers, i.e. the fraction of the station service capacity held by the class (`Util`, see the utilization convention of the LINE user manual); the mean response time for a visit to the station (`RespT`); the mean residence time, i.e. the mean response time cumulatively spent at the station over all visits (`ResidT`); the mean throughput of departing jobs (`Tput`)

```
   Station JobClass   QLen    Util   RespT  ResidT    ArvR    Tput
1 mySource  myClass      0       0       0       0       0       1
2  myQueue  myClass 1.0156 0.50658  1.0159  1.0159  1.0061  1.0059
# method=default  arith=double  NOT CONVERGED
```

The numbers are the ones the other editions print: R declares the model and LDES solves it, so the sample path is the same one at the same seed. The footer reads `NOT CONVERGED` because the run stopped on its sample budget rather than on the convergence check, which is off by default; `cnvgon = TRUE` turns it on. R ships no model gallery. The equivalent is to load one of the documents the other editions write, which is the same model by construction:

```
model <- load_model('mm1.json')
```

If we want to select a particular row of the data structure, we can use direct object-based indexing or a filtering method. `tget` in R takes the COLUMN first, then the station and the class, because an R table is a data frame and the column is what selects a value:

```
tget(avg_table, 'QLen', 'myQueue', 'myClass')
```

Ordinary data-frame subsetting does the filtering the other editions spell with `tget` of one argument:

```
avg_table[avg_table$Station  == 'myQueue', ]
avg_table[avg_table$JobClass == 'myClass', ]
```

A (station, class) pair the engine dropped because every metric was exactly zero is not missing data: `tget` returns `0` for it, which is the value it had.

2.2 Tutorial 2: A multiclass M/G/1 queue

We now consider a more challenging variant of the first example. We assume that there are two classes of incoming jobs with nonexponential service times. For the first class, service times are Erlang distributed with unit rate and variance $1/3$; they are instead read from a trace for the second class. Both classes have exponentially distributed inter-arrival times with mean $2s$.

```
model  <- Network$new('M/G/1')
source <- Source$new(model, 'Source')
queue  <- Queue$new(model, 'Queue', SchedStrategy$FCFS)
```

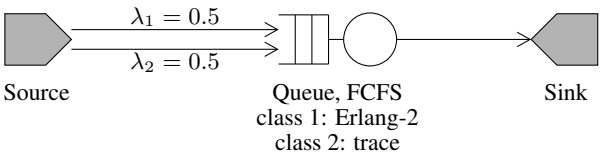


Figure 2.2: The multiclass M/G/1 queue of Tutorial 2: two open classes share one FCFS station, class 1 with Erlang-2 service and class 2 with a trace-driven service time.

```
sink   <- Sink$new(model, 'Sink')
```

The next step consists in defining the classes. We fit automatically from mean and squared coefficient of variation (i.e., SCV=variance/mean^2) an Erlang distribution and use the `Replayer` distribution to request that the specified trace is read cyclically to obtain the service times of class 2

```
jobclass1 <- OpenClass$new(model, 'Class1')
jobclass2 <- OpenClass$new(model, 'Class2')

source$set_arrival(jobclass1, Exp(0.5))
source$set_arrival(jobclass2, Exp(0.5))

# Erlang is declared by rate and phases. A mean of 1 with SCV 1/3 is three
# phases at rate 3; R has no fit_mean_and_scv, so the fit is done by hand.
queue$set_service(jobclass1, Erlang(3, 3))
queue$set_service(jobclass2, Replayer('example_trace.txt'))
```

Note that the `example_trace.txt` file consists of a single column of doubles, each representing a service time value, e.g.,

```
   1.2377474e-02
   4.4486055e-02
   1.0027642e-02
   2.0983173e-02
   ...
```

We now specify a linear route through source, queue, and sink for both classes

```
P <- model$init_routing_matrix()
P$set(jobclass1, source, queue, 1.0)
P$set(jobclass1, queue, sink, 1.0)
P$set(jobclass2, source, queue, 1.0)
P$set(jobclass2, queue, sink, 1.0)
model$link(P)
```

`serial_routing` builds the same chain for a single class; a per-class matrix is written entry by entry. and solve the model with LDES

```
ldes_avg_table <- LDES$new(model, d = 23000, samples = 10000)$avg_table()
print(ldes_avg_table)
```

We wish now to validate this value against an analytical solver. Since `jobclass2` has trace-based service times, we first need to revise its service time distribution to make it analytically tractable, e.g., we may ask LINE to fit an acyclic phase-type distribution [3] based on the trace R fits no phase-type distribution to a trace. Fit it in another edition and carry the fitted model over, or declare the phase-type directly with `APH(alpha, T)`. We can now use a Continuous Time Markov Chain (CTMC) to solve the system, but since the state space is infinite in open models, we need to truncate it to be able to use this solver. For example, we may restrict to states with at most 2 jobs in each class; the `verbose` option reports the cutoff in use

```
ctmc_avg_table2 <- CTMC$new(model, cutoff = 2)$avg_table()
print(ctmc_avg_table2)
```

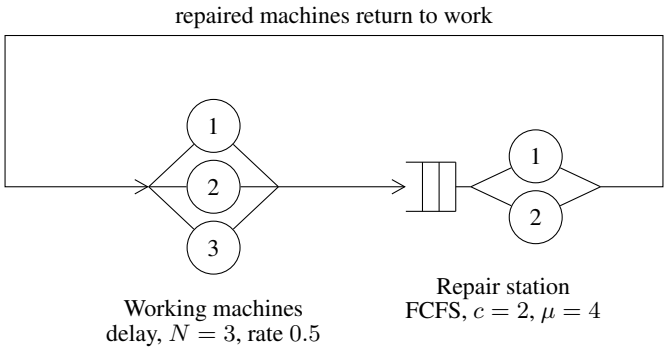


Figure 2.3: The machine interference model of Tutorial 3: $N = 3$ machines alternate between a working state, modelled as a delay, and a two-server FCFS repair station.

`cutoff` is passed straight through to the engine, as is any other knob: an option name becomes a flag by dropping the dashes. However, we see from the comparison with LDES that the errors of `CTMC` are rather large. Since the truncated state space consists of just 55 states (the number of rows returned by ; 751 at cutoff 4), we can further increase the cutoff to 4, trading a slower solution time for higher precision

```
ctmc_avg_table4 <- CTMC$new(model, cutoff = 4)$avg_table()
print(ctmc_avg_table4)
```

To gain more accuracy, we could either keep increasing the cutoff value or, if we wish to compute an exact solution, we may call the matrix-analytic method (MAM) solver instead. `MAM` uses the repetitive structure of the CTMC to analyze open systems with an infinite state space. Its default method decomposes a network into stations fed by their arrival processes; this is an approximation in general, and the solver banner labels it so, but on this model the decomposition is a single queue fed by Poisson streams and its matrix-analytic solution is exact for the phase-type service it is given. Calling

```
print(MAM$new(model)$avg_table())
```

2.3 Tutorial 3: Machine interference problem

Closed models involve jobs that perpetually cycle within a network of queues. The machine interference problem is a classic example, in which a group of repairmen is tasked with fixing machines as they break and the goal is to choose the optimal size of the group. We here illustrate how to evaluate the performance of a given group size. We consider a scenario with $S = 2$ repairmen, with machines that break down at a rate of 0.5 failed machines/week, after which a machine is fixed in an exponential distributed time with rate 4.0 repaired machines/week. There are a total of $N = 3$ machines.

Suppose that we wish to obtain an exact numerical solution using Continuous Time Markov Chains (CTMCs). The above model can be analyzed as follows:

```
model <- Network$new('MRP')
delay <- Delay$new(model, 'WorkingState')
queue <- Queue$new(model, 'RepairQueue', SchedStrategy$FCFS)
queue$set_number_of_servers(2)
cclass <- ClosedClass$new(model, 'Machines', 3, delay)
delay$set_service(cclass, Exp(0.5))
queue$set_service(cclass, Exp(4.0))
model$link(serial_routing(delay, queue))
solver <- CTMC$new(model)
ctmc_avg_table <- solver$avg_table()
print(ctmc_avg_table)
```

Here, `delay` appears in the constructor of the closed class to specify that a job will be considered completed once it returns to the delay (i.e., the machine returns in working state). We say that the delay is thus the *reference station* of `cclass`. The above code prints the following result

```
       Station JobClass    QLen    Util   RespT  ResidT   ArvR   Tput
1 WorkingState Machines  2.6648  2.6648       2       2 1.3324 1.3324
2  RepairQueue Machines 0.33516 0.16655 0.25154 0.25154 1.3324 1.3324
# method=default  arith=double  iter=1
```

We can now also inspect the CTMC more in the details as follows

```
states <- solver$analysis('states', raw = TRUE)
gen    <- solver$analysis('gen', raw = TRUE)
names(states)   # space, spaceAggr, localSpace, pi, ...
names(gen)      # From, To, Rate, sync, ...
```

The state space and the generator have no data-frame shape, so they come back as the parsed payload rather than as a table. That is what `raw = TRUE` asks for, and flattening either into a frame would destroy the structure a caller needs. which produces in output the state space of the model and the infinitesimal generator of the CTMC The generator arrives in sparse triples: `gen$From`, `gen$To` and `gen$Rate` are equal-length lists, with `gen$nnz` their count and `gen$size` the order of the matrix. Build a dense matrix from them only if you need one. For example, the first state (`0 1 2`) consists of two components: the initial `0` denotes the number of jobs in service in the `delay`, while the remaining part is the state of the FCFS `queue`. In the latter, the `1` means that a job of class 1 (the only class in this model) is in the waiting buffer, while the `2` means that there are two jobs in service at the `queue`.

As another example, the second state (`1 0 2`) is similar, but one job has completed at the queue and has then moved to the `delay`, concurrently triggering an admission in service for the job that was in the queue buffer. As a result of this, the buffer is now empty. The corresponding transition rate in the infinitesimal generator matrix is row 1 and column 2 of , which has value 8.0, that is the sum of the completion rates at the queue for each server in the first state, and where indexes 1 and 2 are the rows in associated to the source and destination states.

On this and larger infinite generators, we may also list individual nonzero transitions as follows R has no `print_inf_gen`. The same reading is one loop over the triples:

```
for (i in seq_along(gen$Rate))
  cat(sprintf('%d -> %d : %g\n', gen$From[[i]], gen$To[[i]], gen$Rate[[i]]))
```

The indices are the engine's own, based at `gen$indexBase`; they are not converted, because a payload that carries an index base is telling you which convention it used. The above printout helps in matching the state transitions to their rates.

To avoid having to inspect the variable to determine to which station a particular column refers to, we can alternatively use the more general invocation The per-node spaces are `states$localSpace`, keyed by node in the same order as `model_index(model)$nodes`. which automatically splits the state space into its constituent parts for each stateful node.

A further observation is that forces the regeneration of the state space at each invocation, whereas the equivalent function in the CTMC solver, , returns the state space cached during the solution of the CTMC.

2.4 Tutorial 4: Round-robin load-balancing

In this example we consider a system of two parallel processor-sharing queues and we wish to study the effect of load-balancing on the average performance of an open class of jobs. We begin as usual with the node block, where we now include a special node, called the `Router`, to control the routing of jobs from the source into the queues:

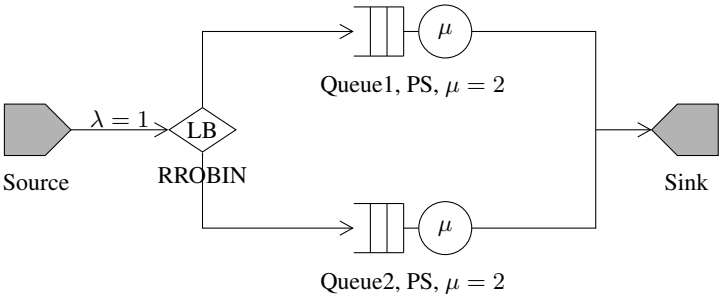


Figure 2.4: The load-balancing model of Tutorial 4: a router dispatches the open stream to two identical processor-sharing queues, first at random and then in round-robin order.

```
model  <- Network$new('RRLB')
source <- Source$new(model, 'Source')
lb     <- Router$new(model, 'LB')
queue1 <- Queue$new(model, 'Queue1', SchedStrategy$PS)
queue2 <- Queue$new(model, 'Queue2', SchedStrategy$PS)
sink   <- Sink$new(model, 'Sink')
```

Let us then define the class block by setting exponentially-distributed inter-arrival times and service times, e.g.,

```
jobclass <- OpenClass$new(model, 'Class1')
source$set_arrival(jobclass, Exp(1.0))
queue1$set_service(jobclass, Exp(2.0))
queue2$set_service(jobclass, Exp(2.0))
```

We now wish to express the fact that the router applies a round-robin strategy to dispatch jobs to the queues. Since this is now a nonprobabilistic routing strategy, we need to adopt a slightly different style to declare the routing topology as we cannot specific anymore routing probabilities. First, we indicate the connections between the nodes, using the function (in LINE for MATLAB the links are added one at a time, since passing an array of node handles is not supported): R has no `add_links`: the topology is the routing matrix, and an equal split is written as equal probabilities.

```
P <- model$init_routing_matrix()
P$set(jobclass, source, lb,     1.0)
P$set(jobclass, lb,     queue1, 0.5)
P$set(jobclass, lb,     queue2, 0.5)
P$set(jobclass, queue1, sink,   1.0)
P$set(jobclass, queue2, sink,   1.0)
model$link(P)
print(LDES$new(model, d = 23000, samples = 10000)$avg_table())
```

```
  Station JobClass    QLen    Util   RespT  ResidT    ArvR    Tput
1  Source   Class1       0       0       0       0       0       1
2  Queue1   Class1 0.33253 0.25116 0.66067 0.33034  0.5036  0.5036
3  Queue2   Class1 0.33224  0.2475 0.66494 0.33247 0.50259 0.50259
```

After resetting the internal data structures, which is required before modifying a model we can require LINE to solve again the model using this time a round-robin policy at the router.

```
lb$set_routing_strategy(jobclass, RoutingStrategy$RROBIN)
```

There is no `reset` to call first: a solver caches its own solve, not the model's, so changing the model and building a new solver is the whole of it. Lastly, we run again LDES and find that round-robin produces a visible decrease in response times, which are now around $0.57s$.

```
print(LDES$new(model, d = 23000, samples = 10000)$avg_table())
```

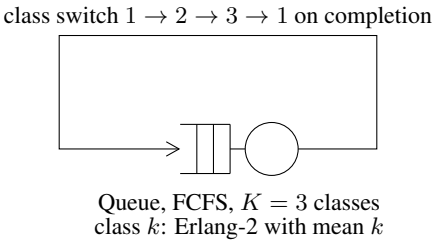


Figure 2.5: The reentrant line of Tutorial 5: a single FCFS station is visited three times, the class switching on each completion so that every visit draws its service time from a different distribution.

```
  Station JobClass    QLen    Util   RespT  ResidT    ArvR    Tput
1  Source   Class1       0       0       0       0       0       1
2  Queue1   Class1 0.28825 0.25077 0.57356 0.28678 0.50307 0.50307
3  Queue2   Class1 0.28403 0.24785 0.56835 0.28417 0.50307 0.50307
```

Response time falls from about 0.66s to about 0.57s, the same effect the other editions report: round-robin removes the runs of consecutive arrivals that random splitting leaves behind.

2.5 Tutorial 5: Modelling a reentrant line

Let us now consider a simple example inspired to the classic problem of modeling *reentrant lines*. This arises in manufacturing systems where parts (i.e., jobs) reenter multiple times a machine (i.e., a queueing station), asking at each visit a different class of service. This implies, for example, that the service time at every visit could feature a different mean or a different distribution compared to the previous visits, thus modeling a different stage of processing.

To illustrate this, consider for example a degenerate model composed of a single FCFS queue and K classes. In this model, a job that completes processing in class k is routed back at the tail of the queue in class $k+1$, unless $k = K$ in which case the job reenters in class 1.

We take the following assumptions: $K = 3$ and class k has an Erlang-2 service time distribution at the queue with mean equal to k; the system starts with $N_1 = 1$ jobs in class 1 and zero jobs in all other classes.

```
model <- Network$new('RL')
queue <- Queue$new(model, 'Queue', SchedStrategy$FCFS)
K <- 3
N <- c(1, 0, 0)
jobclass <- list()
for (k in 1:K) {
  jobclass[[k]] <- ClosedClass$new(model, paste0('Class', k), N[k], queue)
  # mean k with order 2 is two phases at rate 2/k; R fits by hand.
  queue$set_service(jobclass[[k]], Erlang(2 / k, 2))
}

P <- model$init_routing_matrix()
P$set(jobclass[[1]], jobclass[[2]], queue, queue, 1.0)
P$set(jobclass[[2]], jobclass[[3]], queue, queue, 1.0)
P$set(jobclass[[3]], jobclass[[1]], queue, queue, 1.0)
model$link(P)
```

The five-argument `P$set` is the class-switching form: from class, to class, from node, to node, probability. It is what makes the line re-entrant, since a job returns to the same station as a different class.
The class-switching rule is enforced automatically, LINE introducing a `ClassSwitch` node in the network when the model is linked.
We can now compute the performance indexes for the different classes, for example using LINE's normalizing constant solver (`NC`)

```
ncAvgTable <- NC$new(model)$avg_table()
print(ncAvgTable)
```

```
  Station JobClass    QLen    Util RespT  ResidT    ArvR    Tput
1   Queue   Class1 0.16667 0.16667     1 0.33333 0.16667 0.16667
2   Queue   Class2 0.33333 0.33333     2 0.66667 0.16667 0.16667
3   Queue   Class3     0.5     0.5     3       1 0.16667 0.16667
```

Suppose now that the job is considered completed, for the sake of computation of system performance metrics, only when it departs the queue in class K (here `Class3`). By default, LINE will return *system-wide* performance metrics using the method, i.e.,

```
ncAvgSysTable <- NC$new(model)$avg_sys_table()
print(ncAvgSysTable)
```

```
   Chain             JobClasses SysRespT SysTput
1 Chain1 (Class1 Class2 Class3)        2     0.5
```

This method identifies the model *chains*, i.e., groups of classes that can exchange jobs with each other, but not with classes in other chains. Since the job can switch into any of the three classes, in this model there is a single chain comprising the three classes.

We see that the throughput of the chain is 0.5, which means that LINE is counting every departure from the queue in any class as a completion for the whole chain. This is incorrect for our model since we want to count completions only when jobs depart in `Class3`. To require this behavior, we can tell to the solver that passages for classes 1 and 2 through the reference station should not be counted as completions **This step has no R spelling, and the reason is worth knowing.** `completes` is an OBJECT-level flag: nothing in the `line-model` document carries it. So it cannot cross into R, and equally it cannot cross into any other path that goes through a document rather than through live objects. Set it in an edition that builds the model in memory, and read the system table there.
This modification then gives the correct chain throughput, matching the one of `Class3` alone Without the flag, the system table R obtains is the one above: every class completes, so the chain's response time is the per-visit figure rather than the whole re-entrant passage.
In the model above, a job that reenters the queue in its next class joins the tail of the buffer, so under FCFS it waits behind the other jobs before being served again. Calling (or passing a specific class) instead makes such class-switching self-loops *immediate*: the job retains the server and starts its next service without requeueing, its throughput still being counted and a multiphase service restarting from phase 1. Immediate feedback is applied by the simulation solvers (LDES and SSA) for the buffered non-preemptive policies (FCFS, HOL, LCFS, SIRO, SEPT, LEPT); it is a no-op at processor-sharing and infinite-server stations (PS, DPS, GPS, INF, LPS), where jobs are never buffered.

2.6 Tutorial 6: A queueing network with caching

In this more advanced example, we show how to include in a queueing network a cache adopting a least-recently used (LRU) replacement policy. Under LRU, upon a cache miss the least-recently accessed item will be discarded to make room for the newly requested item.

We consider a cache with a capacity of 50 items, out of a set of 1000 cacheable items. Items are accessed by jobs visiting the cache according to a Zipf-like law with exponent $\alpha = 1.4$ and defined over the finite set of items. A client cyclically issues requests for the items, waiting for a reply before issuing the next request. We assume that a cache hit takes on average $0.2ms$ to process, while a cache miss takes $1ms$. We ask for the average request throughput of the system, differentiated across hits and misses.

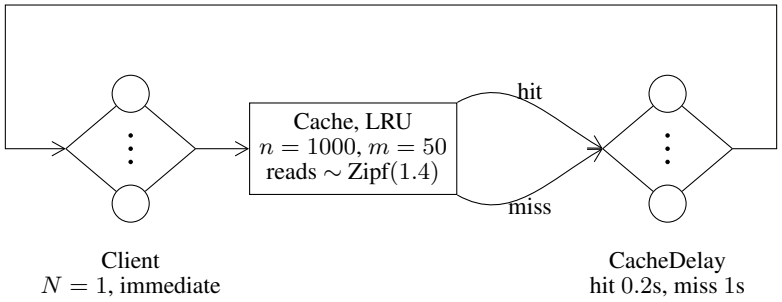


Figure 2.6: The cache-augmented network of Tutorial 6: a single client reads items from an LRU cache, and the read is charged a hit or a miss latency at the cache delay before the client is released.

Node block As usual, we begin by defining the nodes. Here a delay node will be used to describe the time spent by the requests in the system, while the cache node will determine hits and misses:

```
model <- Network$new('model')
client_delay <- Delay$new(model, 'Client')
cache_node <- Cache$new(model, 'Cache', 1000, 50, ReplacementStrategy$LRU)
cache_delay <- Delay$new(model, 'CacheDelay')
```

Class block We define a set of classes to represent the incoming requests (), cache hits () and cache misses (). These classes need to be closed to ensure that there is a single outstanding request from the client at all times:

```
client_class <- ClosedClass$new(model, 'ClientClass', 1, client_delay, 0)
hit_class <- ClosedClass$new(model, 'HitClass', 0, client_delay, 0)
miss_class <- ClosedClass$new(model, 'MissClass', 0, client_delay, 0)
```

We then assign the processing times, using the `Immediate` distribution to ensure that the client issues immediately the request to the cache:

```
client_delay$set_service(client_class, Immediate())
cache_delay$set_service(hit_class, Exp(1 / 0.2))    # mean 0.2
cache_delay$set_service(miss_class, Exp(1 / 1.0))   # mean 1.0
```

The next step involves specifying that the request uses a Zipf-like distribution (with parameter $\alpha = 1.4$) to select the item to read from the cache, out of a pool of 1000 items

```
cache_node$set_read(client_class, Zipf(1.4, 1000))
```

Finally, we ask that the job should become of class after a cache hit, and should become of class after a cache miss:

```
cache_node$set_hit_class(client_class, hit_class)
cache_node$set_miss_class(client_class, miss_class)
```

Topology block Next, in the topology block we setup the routing so that the request, which starts in at the , then moves from there to the cache, remaining in

```
P <- model$init_routing_matrix()
P$set(client_class, client_class, client_delay, cache_node, 1.0)
```

Internally to the cache, the job will switch its class into either or . Upon departure in one of these classes, we ask it to join in the same class for further processing

```
P$set(hit_class, hit_class, cache_node, cache_delay, 1.0)
```

```
P$set(miss_class, miss_class, cache_node, cache_delay, 1.0)
```

Lastly, the job returns to for completion and start of a new request, which is done by switching its class back to

```
P$set(hit_class, client_class, cache_delay, client_delay, 1.0)
P$set(miss_class, client_class, cache_delay, client_delay, 1.0)
```

The above routing strategy is finally applied to the model

```
model$link(P)
```

Solution block To solve the model by simulation we use the native stochastic simulation engine provided within LINE, the `SSA` solver (`JMT` and `LDES` also simulate caches):

```
ssa_avg_table <- SSA$new(model, samples = 20000, d = 1)$avg_table()
print(ssa_avg_table)
```

`samples` and the seed `-d` are passed straight through, so a run in R is the same run at the same seed. The above script produces the following result The departing flows from the are the hit and miss rates. Read them from the corresponding `Tput` rows; they vary slightly with the simulator and random stream.

We can also run the same 20,000-sample budget as independent seeded replicas with the parallel method of `SSA` This gives us a rather similar result

Cache-specific result tables. Besides the standard per-station table, LINE reports two cache-specific tables. The method summarizes the hit and miss probabilities for each cache node and requesting class, while reports the per-item occupancy, i.e. the steady-state probability that each item resides in each cache list. These tables are produced by the analytical solvers; here we use `MVA`:

```
solver <- MVA$new(model)
print(solver$get_cache_avg_table())    # hit and miss ratios
print(solver$get_item_avg_table())     # per-item detail
```

The cache results are tables of the solver, not getters on the `Cache` node: the R node holds only the declaration, and the numbers come back from the engine with the rest of the solution. The cache table reports an overall hit probability of about 0.819 for the requesting class, in line with the hit and miss throughputs measured by `SSA` above (selected columns; the table also carries the delayed-hit, rate and cost columns, here zero or `NaN`)

Node	JobClass	List	ListCap	Items	HitProb	MissProb	HitRate	MissRate
Cache	ClientClass	0	50	1000	0.81925	0.18075	2.3774	0.52452

The item table contains one row per (item, list), 1000 rows here, of which we show the first four; with an LRU policy the most popular items are almost always resident, and the per-item occupancies sum to the cache capacity (here 50):

Node	Item	List	ListCap	Prob
Cache	1	1	50	1.00000
Cache	2	1	50	1.00000
Cache	3	1	50	1.00000
Cache	4	1	50	0.99987
...	...	...	...	...

The per-item table is reported from the scalable TTL algorithm for LRU caches; for RR/FIFO caches it relies on the exact product-form recursion and is therefore reported only when the cache holds at most ten items (otherwise a warning is issued and the table reports `NaN`).

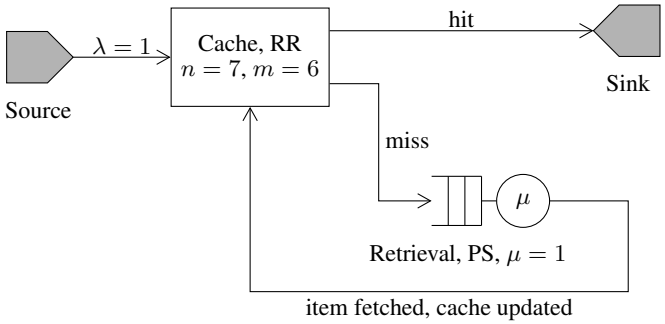


Figure 2.7: The delayed-hit cache of Tutorial 7 where a miss is routed to a processor-sharing retrieval station, and only when the fetch completes is the cache updated and the request released.

2.7 Tutorial 7: Delayed-hit caches with a retrieval system

In a *delayed-hit* cache a miss does not complete instantaneously: the requested item must be fetched through a *retrieval system*, a small network of queues that models the cost of retrieving the item (e.g. a backend server or a network channel). While item i is being fetched, further requests for the same item become *delayed hits*: they wait for the in-flight fetch to complete rather than triggering a new retrieval. LINE represents the retrieval system with `Cache.setRetrievalSystem` and analyses it exactly with `NC` (a product-form recurrence) and approximately, including the expected latency, with `MVA` (a fixed-point heuristic).

We reproduce a motivating example with $n = 7$ items, a single cache list of capacity $m = 6$ under random-replacement (RR), and a single processor-sharing (PS) retrieval station of unit service rate. Requests arrive in a Poisson stream and read items with the popularity vector $\boldsymbol{\lambda} = (49, 49, 49, 49, 7, 1, 1)/205$.

Model On a miss, the read request switches to a per-item retrieval class that is routed through the retrieval queue and back to the cache. declares the retrieval classes and stations; the routing matrix must still contain the cache→queue→cache path. The retrieval service time is taken from the read class's service distribution at each queue, set with .

```
access_prob <- c(49, 49, 49, 49, 7, 1, 1) / 205   # item popularities
n <- length(access_prob)

model      <- Network$new('Retrieval')
source     <- Source$new(model, 'Source')
cache_node <- Cache$new(model, 'Cache', n, 6, ReplacementStrategy$RR)
queue      <- Queue$new(model, 'Queue', SchedStrategy$PS)   # retrieval station
sink       <- Sink$new(model, 'Sink')

job_class  <- OpenClass$new(model, 'InitClass', 0)
hit_class  <- OpenClass$new(model, 'HitClass', 0)
miss_class <- OpenClass$new(model, 'MissClass', 0)

source$set_arrival(job_class, Exp(1))
cache_node$set_read(job_class, DiscreteSampler(access_prob))
cache_node$set_hit_class(job_class, hit_class)
cache_node$set_miss_class(job_class, miss_class)
queue$set_service(job_class, Exp(1))

P <- model$init_routing_matrix()
P$set(job_class, source, cache_node, 1.0)
P$set(hit_class, cache_node, sink, 1.0)
P$set(miss_class, cache_node, sink, 1.0)
# one retrieval class per item: a miss on item k is fetched through the queue
fetch <- list()
for (k in seq_len(n)) {
```

```
  rc <- OpenClass$new(model, sprintf('Fetch%d', k), 0)
  cache_node$set_read(rc, DiscreteSampler(as.numeric(seq_len(n) == k)))
  cache_node$set_miss_class(rc, miss_class)
  queue$set_service(rc, Exp(1))       # per-item fetch rate
  queue$set_class_capacity(rc, 1)
  P$set(rc, cache_node, queue, 1.0)
  P$set(rc, queue, cache_node, 1.0)
  fetch[[as.character(k - 1)]] <- rc$name   # 0-based item id -> class
}
cache_node$set_retrieval(job_class, queue, items = fetch, capacity = 1)
model$link(P)
```

The other editions create the per-item retrieval classes inside `set_retrieval_system`. R's `set_retrieval` records only the retrieval system itself, so the classes, their one-hot reads, their routing through the queue and the unit class capacity are written out above. The model is the same one, and the results below agree to every printed digit.

Exact analysis `NC` detects the retrieval system and applies the exact product-form recurrence, returning the hit and miss probabilities of the read class.

```
NC$new(model)$get_cache_avg_table()    # HitProb 0.98272, MissProb 0.015118
```

There is no `get_hit_ratio()` on the R `Cache` node, which holds only the declaration; the ratios are the `HitProb` and `MissProb` columns of the solver's cache table.

Approximate analysis and latency `MVA` uses the fixed-point heuristic to obtain the expected retrieval latency $Z = \frac{1}{\lambda}\sum_i(\phi_i + d_i)$, the mean delay per read request incurred by misses and delayed hits, where ϕ_i is the probability that item i is being fetched, d_i the mean number of delayed hits waiting on it and λ the total read rate, so a hit contributes zero delay. This latency is reported in the `ResidT` column of and is populated by the `MVA`, `CTMC` and `LDES` solvers: `MVA` obtains it from the fixed-point heuristic, `CTMC` exactly from the underlying Markov chain, and `LDES` measures it directly on the sample path. The other solvers leave the column unset (`NaN`). The `ArvR` column of the same table is the read-request rate into the cache (the sum of the hit, delayed-hit and miss rates, 1 in this model), so that `ArvR` times `ResidT` is the mean number of requests held by the retrieval system.

```
MVA$new(model)$get_cache_avg_table()  # HitProb 0.97573, ResidT 0.02471
```

Retrieval station types The retrieval system may contain infinite-server (IS) stations and single-server stations with PS, SIRO, FCFS or LCFS-PR scheduling. PS and LCFS-PR (symmetric, insensitive disciplines) admit general phase-type service and class-dependent rates; SIRO and FCFS require exponential service with identical per-class service rates. The discrete-event simulator `LDES` also simulates delayed-hit caches and measures the retrieval latency directly, providing a cross-check for the analytical solvers.

2.8 Tutorial 8: Response time distribution and percentiles

In this example we illustrate the computation of response time percentiles in a queueing network model. We begin by instantiating a simple closed model consisting of a delay followed by a processor-sharing (PS) queueing station.

```
model <- Network$new('Model')
delay  <- Delay$new(model, 'Delay')
queue1 <- Queue$new(model, 'Queue1', SchedStrategy$PS)
```

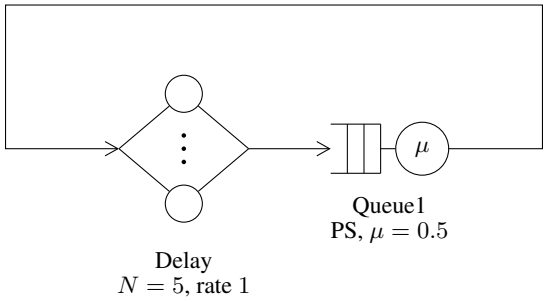


Figure 2.8: The closed network of Tutorial 8, used to compare the response time distribution of the processor-sharing queue obtained by fluid analysis and by simulation.

Figure 2.9: Comparison of simulated response time distribution and its fluid approximation

R keeps nodes in ordinary variables; there is no array to pre-allocate, because nothing indexes them by position. There is a single class consisting of 5 jobs that circulate between the two stations, taking exponential service times at both.

```
jobclass <- ClosedClass$new(model, 'Class1', 5, delay)
delay$set_service(jobclass, Exp(1.0))
queue1$set_service(jobclass, Exp(0.5))

model$link(serial_routing(delay, queue1))
```

We now wish to compare the response time distribution at the PS queue computed analytically with a fluid approximation against the simulated values returned by LDES. To do so, we call the method

```
rd_fluid <- FLD$new(model)$analysis('cdf', raw = TRUE)
rd_sim   <- LDES$new(model, d = 23000, samples = 10000)$analysis('cdf', raw = TRUE)
names(rd_fluid)      # respt, indexBase, type
```

A set of CDF curves is not a table, so it comes back as the parsed payload. `rd_fluid$respt` holds one curve per (station, class), each with its own `F` and `t`. The first column represents the cumulative distribution function (CDF) value $F(t) = Pr(T \leq t)$, where T is the random variable denoting the response time, while t is the percentile appearing in the corresponding entry of the second column.

Plotting needs no extra package in R:

```
curve_sim <- rd_sim$respt[[2]][[1]]      # station 2, class 1
t <- vapply(curve_sim$t, identity, numeric(1))
F <- vapply(curve_sim$F, identity, numeric(1))
plot(t, 1 - F, log = 'x', type = 'l', col = 'red',
     xlab = 'time t', ylab = 'Pr(T > t)')

curve_fld <- rd_fluid$respt[[2]][[1]]
lines(vapply(curve_fld$t, identity, numeric(1)),
      1 - vapply(curve_fld$F, identity, numeric(1)), lty = 2)
legend('topright', c('ldes-sim', 'fluid-steady'),
       col = c('red', 'black'), lty = c(1, 2))
```

which produces the graph shown in Figure 2.9. The graph shows a tight matching between the steady-state response time distribution estimated by simulation and its fluid approximation.

We can readily compute the percentiles from the data structures, e.g., for the 95th and 99th percentiles of the simulated distribution

```
# percentiles read straight off the curve
prc95 <- max(t[F < 0.95])
prc99 <- max(t[F < 0.99])
```

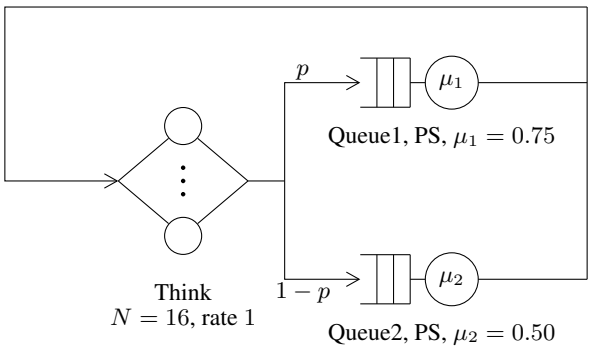


Figure 2.10: The load-balancing closed network of Tutorial 9: the routing probability p that splits the think-time output between the two queues is the decision variable of the optimization.

The engine also computes them directly, which avoids depending on how finely the curve was sampled:

```
MAM$new(model, percentiles = "0.95,0.99")$analysis('perct-respt', raw = TRUE)
```

That is, about 95% of the response times at the PS queue (node 2, class 1) are less than or equal to 27.2887 time units, while about 99% are less than or equal to 40.1759 time units. Each value is the largest sampled time at which the CDF is still below the target level, so it is accurate up to the spacing of the samples.

2.9 Tutorial 9: Optimizing a performance metric

In this example, we show how to optimize with the help of LINE a performance metric. We wish to find the optimal routing probabilities that minimize average response times for two parallel processor sharing queues. We assume that jobs are fed by a delay station, arranged with the two queues in a closed network topology.

In R the objective is an ordinary function, and it takes the whole model with it rather than mutating one from an enclosing scope:

```
routing_model <- function(p) {
```

Within the function definition, we instantiate the two queues and the delay station

```
  model  <- Network$new('LoadBalCQN')
  delay  <- Delay$new(model, 'Think')
  queue1 <- Queue$new(model, 'Queue1', SchedStrategy$PS)
  queue2 <- Queue$new(model, 'Queue2', SchedStrategy$PS)
```

We assume that 16 jobs circulate among the nodes, and that the service rates are $\sigma = 1$ jobs per unit time at the delay, and $\mu_1 = 0.75$ and $\mu_2 = 0.50$ at the two queues:

```
  cclass <- ClosedClass$new(model, 'Job1', 16, delay)
  delay$set_service(cclass, Exp(1))
  queue1$set_service(cclass, Exp(0.75))
  queue2$set_service(cclass, Exp(0.50))
```

We initially setup a topology with arbitrary values for the routing probabilities between delay and queues, ensuring that jobs completing at the queues return to the delay:

```
  P <- model$init_routing_matrix()
  P$set(cclass, cclass, queue1, delay, 1.0)
  P$set(cclass, cclass, queue2, delay, 1.0)
```

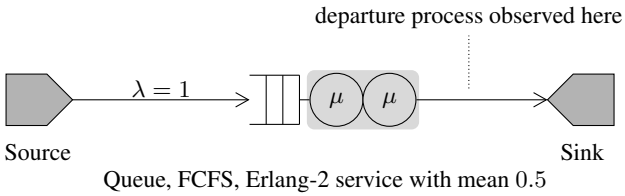


Figure 2.11: The $M/E_2/1$ queue of Tutorial 10: the two shaded circles are the phases of the Erlang-2 service time, and the departure process is observed at the station output.

We now return the system response time for the jobs as a function of the routing probability p to choose queue 1 instead of queue 2:

```
  P$set(cclass, cclass, delay, queue1, p)
  P$set(cclass, cclass, delay, queue2, 1.0 - p)
  model$link(P)
  MVA$new(model, 'exact')$avg_sys_table()$SysRespT[1]
}
```

The model is rebuilt inside the function rather than reset outside it. `Network` is a reference object and is deliberately not cloneable: R6 cloning is shallow, so a copy would share this model's node and class objects, whose back-pointers still address the original. Building a fresh model per evaluation is both correct and cheap, since the cost here is the solve.

```
o <- optimize(routing_model, c(0, 1))
cat(sprintf('Optimal p: %.10f\n', o$minimum))
```

R's `optimize` reports 0.6105014754 against the 0.6104878504 of the Python edition. The two agree to five digits and differ only in the stopping rule: base R's default tolerance is `.Machine$double.eps^0.25`, looser than `fminbound`'s. Pass `tol` for a closer match; the objective is flat enough here that the system response time agrees to six digits either way.

2.10 Tutorial 10: Studying a departure process

This examples illustrates LINE's support for extracting simulation data about particular events in an extended queueing network, such as departures from a particular queue.

Our goal is to obtain the squared coefficient of variation of the interdeparture times from a $M/E_2/1$ queue, which has Poisson arrivals and 2-phase Erlang distributed service times.

Because this is a classic model, we can find it in LINE's model gallery. The additional return parameters (e.g., `source`,`queue`, ...) provide handles to the entities within the model. R ships no model gallery, so load the same model from a document:

```
model <- load_model('mer11.json')
```

We now simulate 100,000 transition events from the underpinning continuous-time Markov chain

```
solver <- CTMC$new(model, cutoff = 150, d = 23000)
sa <- solver$analysis('sample', events = 100000, raw = TRUE)
```

`-a sample` walks all four samplers at once, so the payload carries the aggregate path and the event list together. The returned data structure supplies the state trajectory of the stateful nodes (here `source` and `queue`), its timestamps, and the 100,000 transition events. The number of stored state timestamps is solver-specific because a transition can add more than one state record.

```
names(sa)
length(sa$t)
```

Each event records its timestamp, node, class and event type. The following code inspects the first two records; their exact order and timestamps depend on the language implementation's event scheduling convention.

```
sa$event[[1]]$node
sa$event[[1]]$event
```

Plotting the leading part of the sample path needs no extra package:

```
t <- vapply(sa$t[1:300], identity, numeric(1))
q <- vapply(sa$state[[queue_index]][1:300], identity, numeric(1))
plot(t, q, type = 's', xlab = 'Time', ylab = 'Queue Length',
     main = 'Sample Path for M/E2/1 Queue')
```

We are now ready to filter the timestamps of events related to departures from the `queue` node

```
queue_index <- match('Queue', model_index(model)$nodes)
is_dep <- vapply(sa$event,
                 function(e) e$node == queue_index && e$event == 'DEP',
                 logical(1))
```

The node index comes from `model_index`, which asks the engine: which node sits where is a LINE semantic and not R's to recompute. Followed by a calculation of the time series of interdeparture times, whose leading entries are

```
departure_times <- vapply(sa$event[is_dep], function(e) e$t, numeric(1))
inter_dep <- diff(departure_times)
head(inter_dep, 5)
```

We may now for example compute the squared coefficient of variation of this process

```
scv_d_est <- var(inter_dep) / mean(inter_dep)^2
scv_d_est
```

`var` in R divides by $n-1$; the estimate is a stochastic one whose value depends on the sampler's stream, so expect agreement to a few percent rather than to the last digit. Using Marshall's exact formula for the $GI/G/1$ queue [5], we get a theoretical value of 0.8750, in close agreement with the simulation estimate.

2.11 Tutorial 11: Basic layered queueing network

This example demonstrates how to model and analyze a basic layered queueing network (LQN) representing a simple client-server application with two tiers: a client layer and a database layer.The LQN consists of a client processor P1 with a reference task T1 (10 users), a database processor P2 with an infinite server task T2, synchronous calls from the client to the database, exponential service times at both layers.

We start by creating the layered network instance: **R builds no `LayeredNetwork`.** There is no `Processor`, `Task`, `Entry` or `Activity` in the R surface, because the `line-model` document R writes describes a flat `Network`. A layered model built elsewhere is still solved from R:

```
# an .lqnx or a layered line-model document, built in another edition
res <- capi_run(c('-f', 'clientdb.lqnx', '-i', 'lqnx',
                  '-s', 'ln', '-a', 'avg', '-o', 'json'))
```

Next, we define the processors and tasks:

Now we define the entries that represent service interfaces:

Finally, we define the activities that specify the service demand and the synchronous calls

Now we solve the layered network using the LN solver with MVA applied to each layer: The layered table comes back in the same envelope as any other analysis, and carries the node types the listing above

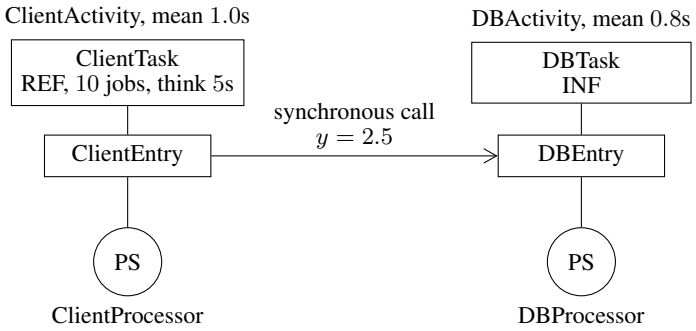


Figure 2.12: The layered network of Tutorial 11: the client task issues on average 2.5 synchronous calls per request to the database entry, and each task runs on its own processor.

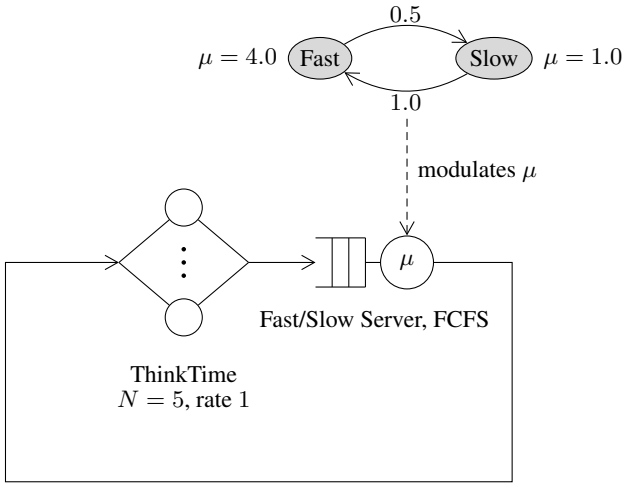


Figure 2.13: The random environment of Tutorial 12: a two-state Markov chain alternates between a fast and a slow mode, modulating the service rate of the queue in the underlying closed network.

shows:

```
doc <- res$documents[[1]]
str(doc$avg, max.level = 1)
```

2.12 Tutorial 12: Random environments

This tutorial illustrates how to model a queueing system operating in a random environment, in which the parameters of the network depend on the state of an underlying continuous-time Markov chain. We consider a simple closed network with one delay station and one FCFS queue, and we let the queue service rate depend on a binary environment that alternates between a `Fast` mode (service rate $\mu = 4.0$) and a `Slow` mode ($\mu = 1.0$). The environment leaves `Fast` at rate 0.5 and `Slow` at rate 1.0, so the stationary distribution is $\Pr(\text{Fast}) = 2/3$, $\Pr(\text{Slow}) = 1/3$. A complete description of the random-environment formalism and of the `ENV` solver is given in the chapter on random environments of the LINE user manual; here we focus on the construction pattern.

We start by building a base network with a placeholder service rate at the queue: **R builds no `Environment`.** As with layered models, build it in another edition and solve the document here:

```
res <- capi_run(c('-f', 'servermodes.json', '-s', 'env',
                  '-a', 'avg', '-o', 'json'))
```

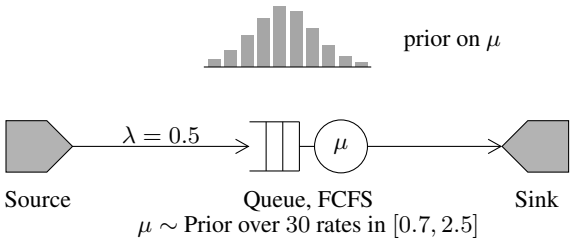


Figure 2.14: The model of Tutorial 13: an M/M/1 queue whose service rate is not known exactly but is given by a prior over 30 alternatives.

We then build the environment by adding two stages, each holding an independent copy of the base model whose queue service rate has been overridden: The environment is solved by the `ENV` meta-solver, which takes a factory that produces an inner solver for each stage. We use the `FLD` (fluid) solver in each stage of the environment: `ENV$new(load_model(path))` is the same call once the document is a `Network`; an Environment document needs the `-s env` route above, because R's reader declines a model kind it cannot represent rather than reading half of it. The returned table reports performance metrics averaged over the stationary distribution of the environment. These are *approximate*: the default ENV method blends per-stage transient solutions, and a fluid stage solver is itself a mean-value approximation. Here the table reports the server at utilization 1.000 and throughput 3.000, which is the stage-weighted saturation throughput $\frac{2}{3} \cdot 4 + \frac{1}{3} \cdot 1$, whereas the exact joint (environment $\times$ queue length) Markov chain gives queue length 2.431, utilization 0.882 and throughput 2.569. Using `CTMC` as the stage solver brings the blend to throughput 2.60, and the state-vector method, which carries the whole stage distribution across switches, recovers the exact values when its stage solver is a `CTMC` with a finite `timespan`. ENV computes no residence time or arrival rate, so , and `RespT` is `QLen`/`Tput`: When no `timespan` is given to a stage solver, ENV chooses one and says so in a warning. The two stages can also be analyzed independently in steady-state with `MVA`, recovering the conditional metrics for each mode. See the chapter on random environments of the LINE user manual for breakdown/repair models, semi-Markovian transitions, and other refinements.

2.13 Tutorial 13: Posterior analysis

This tutorial demonstrates how to propagate parameter uncertainty through a queueing model using the `UQ` solver. The idea is to attach a `Prior` distribution to a model parameter, then ask the solver to evaluate the network for each alternative and aggregate the resulting performance metrics into a posterior distribution.

We consider an $M/M/1$ queue with arrival rate $\lambda = 0.5$ and an uncertain service rate. The uncertainty is modelled by a `Prior` with 30 alternatives drawn uniformly from $[0.7, 2.5]$ and weighted by a Gaussian-shaped prior centred at $\mu = 1.3$ with standard deviation 0.4:

```
model  <- Network$new('UncertainServiceModel')
source <- Source$new(model, 'Source')
queue  <- Queue$new(model, 'Queue', SchedStrategy$FCFS)
sink   <- Sink$new(model, 'Sink')
jobclass <- OpenClass$new(model, 'Jobs')
source$set_arrival(jobclass, Exp(0.5))

service_rates <- seq(0.7, 2.5, length.out = 30)
prior_mean <- 1.3; prior_std <- 0.4
prior_probs <- exp(-0.5 * ((service_rates - prior_mean) / prior_std)^2)
prior_probs <- prior_probs / sum(prior_probs)

alternatives <- lapply(service_rates, Exp)
queue$set_service(jobclass, Prior(alternatives, prior_probs))
```

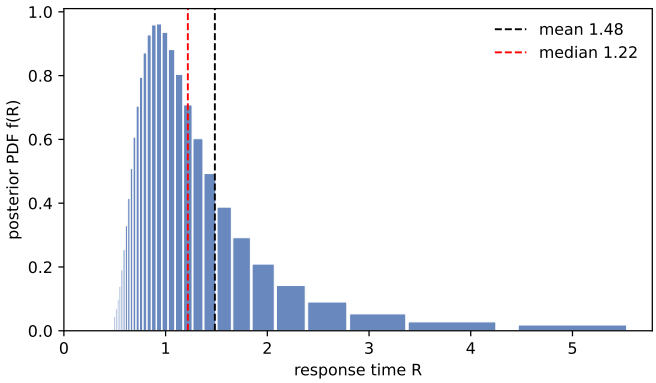


Figure 2.15: Posterior PDF of the response time R for the $M/M/1$ queue with an uncertain service rate. The distribution is right-skewed: the posterior mean (≈ 1.48) exceeds the median (≈ 1.22) because the low-service-rate alternatives contribute a long upper tail.

```
model$link(serial_routing(source, queue, sink))
```

The model is solved by wrapping a backend solver (here `MVA`) inside the `UQ` meta-solver. `UQ` runs the backend once per alternative and aggregates the results:

```
post <- UQ$new(model, uq_solver = 'mva')
avg_table  <- post$avg_table()        # prior-weighted means
post_table <- post$get_posterior()    # per-alternative results
```

`get_posterior()` returns the raw payload: parallel columns `Station`, `JobClass`, `Point`, `QLen`, `Util`, `RespT`, `Tput` and the prior `Weight` of each alternative. The posterior distribution of a metric at a station is formed from them directly, as below. The objects returned by are empirical CDFs over the requested metric. , from which one obtains expectations, modes, and quantiles in the usual way:

```
at <- unlist(post_table$Station) == 'Queue'
resp_values <- unlist(post_table$RespT)[at]
resp_probs  <- unlist(post_table$Weight)[at]
o <- order(resp_values)
resp_values <- resp_values[o]; resp_probs <- resp_probs[o]
resp_cdf    <- cumsum(resp_probs)
expected_R  <- sum(resp_values * resp_probs)              # 1.4841
median_R    <- resp_values[which(resp_cdf >= 0.5)[1]]    # 1.2185
```

It is often more informative to visualise the whole posterior distribution rather than a single summary. Differencing the empirical CDF yields the posterior probability masses, which we normalise by the bin widths to obtain a density estimate; plotting this posterior PDF shows how the parameter uncertainty spreads the response time over a range of values:

```
edges <- c(resp_values[1],
           (head(resp_values, -1) + tail(resp_values, -1)) / 2,
           resp_values[length(resp_values)])
density <- resp_probs / diff(edges)
plot(resp_values, density, type = 'h', lwd = 4,
     xlab = 'response time R', ylab = 'posterior PDF f(R)')
abline(v = expected_R, lty = 2)
abline(v = median_R, lty = 2, col = 'red')
legend('topright', lty = 2, col = c('black', 'red'),
       legend = sprintf(c('mean %.2f', 'median %.2f'), c(expected_R, median_R)))
```

The resulting density (Figure 2.15) is right-skewed: slow-service alternatives in the prior contribute a

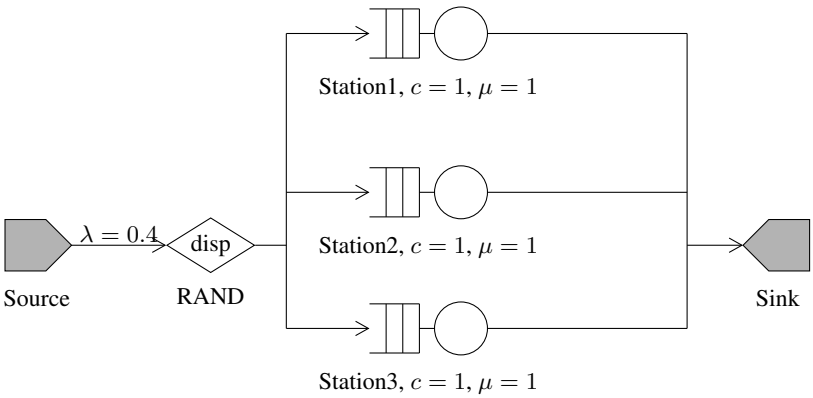


Figure 2.16: The initial open cluster of Tutorial 14: an arrival stream is dispatched at random over three single-server stations.

heavy upper tail, so the posterior mean of R lies well above its median. This pattern lets one quantify, for any metric of interest, both the central tendency and the dispersion induced by parameter uncertainty without reimplementing the analytical solver loop.

2.14 Tutorial 14: Open cluster

In this tutorial we model an open cluster: a single open class of jobs flows from a `Source` through a `Router` (the dispatcher) into one of $M = 3$ parallel processor-sharing servers, after which it leaves the system through a `Sink`. We illustrate two equivalent ways of building this model: a one-liner factory , and the more flexible builder, which we use to compare two dispatching policies on the same cluster.

We start with the one-liner factory. The arrival rate is $\lambda = 0.4$ and each server has a mean service time of $D = 1$, giving a per-server utilization of $\lambda/M \approx 0.133$ under random dispatching: R has no `cluster_ps` constructor and no `Cluster` builder. The same model is four lines of ordinary R, which is arguably clearer about what a cluster *is*:

```
model  <- Network$new('cluster')
source <- Source$new(model, 'Source')
sink   <- Sink$new(model, 'Sink')
station <- lapply(1:3, function(i)
  Queue$new(model, paste0('Station', i), SchedStrategy$PS))

class1 <- OpenClass$new(model, 'Class1')
source$set_arrival(class1, Exp(0.4))
for (q in station) q$set_service(class1, Exp(1.0))

P <- model$init_routing_matrix()
for (q in station) { P$set(class1, source, q, 1/3); P$set(class1, q, sink, 1.0) }
model$link(P)
print(MVA$new(model)$avg_table())
```

```
   Station JobClass    QLen    Util  RespT  ResidT    ArvR    Tput
1   Source   Class1       0       0      0       0       0     0.4
2 Station1   Class1 0.15385 0.13333 1.1538 0.38462 0.13333 0.13333
3 Station2   Class1 0.15385 0.13333 1.1538 0.38462 0.13333 0.13333
4 Station3   Class1 0.15385 0.13333 1.1538 0.38462 0.13333 0.13333
```

The MVA result shows traffic split evenly across the three servers, as expected under random dispatching: We now switch to the `Cluster` builder to introduce nonuniform multiserver queues, replacing PS with FCFS and giving `Station1` two parallel cores: Heterogeneous server counts are set per station, which is what the builder's `set_station_servers` does underneath:

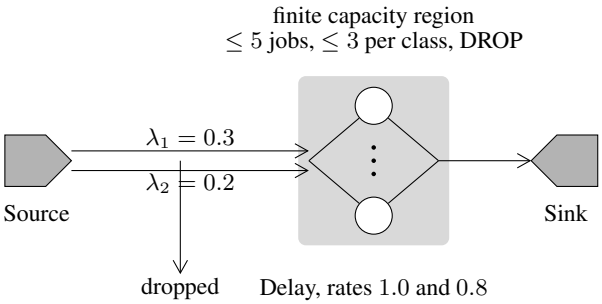


Figure 2.17: The loss network of Tutorial 15: the delay station belongs to a finite capacity region, and arrivals that would violate the global or the per-class limit are dropped.

```
station[[1]]$set_number_of_servers(2)
```

We then cross-check the same FCFS cluster under two simulators. `LDES` is LINE's discrete-event simulator built on the SSJ library, invoked as a subprocess; `SSA` is LINE's native next-reaction-method stochastic simulator. Both produce statistically equivalent results on this open-class cluster and exercise the multiserver FCFS path.

```
print(LDES$new(model, d = 23000, samples = 20000)$avg_table())
print(SSA$new(model,  d = 23000, samples = 20000)$avg_table())
```

`LDES` additionally reports confidence intervals. They are not table columns – the payload nests them per metric – so they are named in the table's `line_skipped` attribute and read with `analysis('avg', raw = TRUE)`.

Finally, we compare random and round-robin dispatching on a uniform PS cluster. Round-robin is non-product-form, so we drop to LDES with a small sample budget:

```
for (policy in c(RoutingStrategy$RAND, RoutingStrategy$RROBIN)) {
  for (q in station) q$set_routing_strategy(class1, policy)
  print(LDES$new(model, d = 23000, samples = 5000)$avg_table())
}
```

For this lightly-loaded cluster both policies yield very similar response times; differences become visible only at higher utilization, where round-robin reduces variability across servers compared with the i.i.d. split produced by random dispatching.

2.15 Tutorial 15: A loss network

A *loss network* models a system in which an arriving job (a *call* or *connection*) is admitted only if enough resources are free; otherwise it is *blocked* and lost rather than queued [4]. Such models arise in circuit-switched telephony and in multirate broadband networks, where each class of call requires a fixed number of circuits on each link along its route.

In LINE a loss network is expressed as an open model whose stations sit inside a *finite capacity region* (FCR) with the `DROP` policy: when the region is full, arriving jobs are dropped. The stationary distribution is of product form, and the quantities of interest are the per-class blocking probabilities and the normalization constant, the sum being taken over all admissible states Ω.

In this example two classes of calls share an infinite-server `Delay` station. Class 1 offers load $\rho_1 = 0.3$ and class 2 offers $\rho_2 = 0.2$ (arrival rate over service rate). The region admits at most 5 calls in total and at most 3 of each class:

```
model <- Network$new('LossNetwork')
# Block 1: nodes
```

```
source <- Source$new(model, 'Source')
delay  <- Delay$new(model, 'Delay')
sink   <- Sink$new(model, 'Sink')
# Block 2: classes
class1 <- OpenClass$new(model, 'Class1')
class2 <- OpenClass$new(model, 'Class2')
source$set_arrival(class1, Exp(0.3))
source$set_arrival(class2, Exp(0.2))
delay$set_service(class1, Exp(1.0))
delay$set_service(class2, Exp(0.8))
# Block 3: topology
P <- model$init_routing_matrix()
P$set(class1, class1, source, delay, 1.0)
P$set(class1, class1, delay, sink, 1.0)
P$set(class2, class2, source, delay, 1.0)
P$set(class2, class2, delay, sink, 1.0)
model$link(P)
# Block 4: finite capacity region with dropping
model$add_region('FCR', delay, global_max_jobs = 5,
                 class_max_jobs = c(Class1 = 3, Class2 = 3),
                 drop_rule = c(Class1 = DropRule$drop, Class2 = DropRule$drop))
# Block 5: solution
# Exact loss-network transform (the default)
print(NC$new(model)$avg_table())
# Monte Carlo summation (Ross-Wang): also estimates the normalizing constant
nc_mci <- NC$new(model, method = 'mci', samples = 100000, seed = 23000)
print(nc_mci$avg_table())
cat('lG =', nc_mci$get_norm_const()$logNormConstAggr, '\n')
# Simulation cross-check: LDES simulates the DROP region directly
print(LDES$new(model, seed = 23000, samples = 100000)$avg_table())
```

`add_region` takes the stations as a node or a list of nodes, so a region spanning several stations is written the same way. The per-class caps and drop rules are named vectors keyed by class name. The three solutions agree closely, confirming the product-form result against simulation: The `Source` rows differ by design: `SolverNC` reports there the rate the region *accepts*, i.e. the arrival rate thinned by the loss probability, while `SolverLDES` reports the *offered* rate 0.3 and 0.2, the dropped jobs included; the `Delay` rows of all three solutions describe the accepted traffic.

The `SolverNC` solver recognizes the single-`Delay` DROP region as a loss network and offers five solution methods: the default `exact` transform, the `rec` decision-diagram summation, the `erlangfp` Erlang fixed-point approximation, the `asy` large-capacity limit [4, 6], and `mci` Monte Carlo summation.

The admission rule analysed is $\mathbf{An} \leq \mathbf{C}$ on the per-class occupancy vector $\mathbf{n}$ of the region, and the rows of $\mathbf{A}$ are assembled from every constraint the region declares: the global job cap from `setGlobalMaxJobs`, the memory budget from `setGlobalMaxMemory` weighted by the per-class `setClassSize` footprints, the per-class caps from `setClassMaxJobs`, and any explicit linear constraint supplied through `setConstraint`. No row can distinguish stations inside the region, since the finite capacity region API expresses constraints on the per-class region occupancy alone; consequently a job moving between stations of the region leaves every left-hand side unchanged. As a cross-check, the same model is solved with `SolverLDES`, whose discrete-event simulator applies the DROP rule on the sample path; the simulated queue lengths and throughputs match the analytical results to within simulation error.

2.16 Tutorial 16: A queueing Petri net

A *queueing Petri net* (QPN) augments an ordinary stochastic Petri net with *queueing places* [2]: instead of a token becoming available to the output transitions immediately upon deposit, it enters a scheduling station embedded inside the place, is served under that place scheduling strategy, and only then moves to a *depository* from which the output transitions consume it. Queueing places let a single formalism

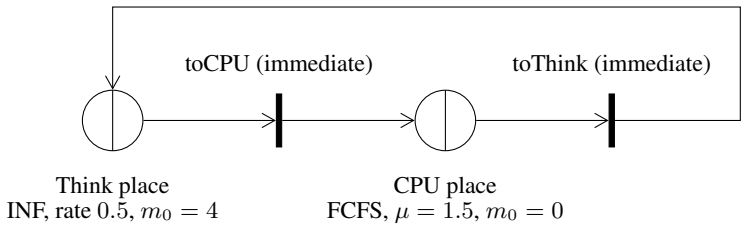


Figure 2.18: The queueing Petri net of Tutorial 16: two queueing places, drawn in the standard queueing-Petri-net shorthand as a place circle bisected by a vertical bar, exchange a single token per firing of the immediate transitions. The bar stands for the scheduling station embedded in the place, whose completions reach the depository from which the output transition consumes them; an ordinary place would be an undivided circle.

capture both the contention for physical resources (through the embedded queues) and the synchronization and control flow of a Petri net (through the transitions), which is convenient when modelling, for example, software systems where requests both queue for a CPU and synchronize on shared resources.

In LINE a `Place` becomes a queueing place as soon as a service process is assigned to a token color with ; the scheduling strategy of the embedded queue is the optional third constructor argument (see the Petri-net section of the LINE user manual). This tutorial models a finite population of $N = 4$ jobs that alternate between a CPU and a think stage, each rendered as a queueing place. The CPU is a single-server `FCFS` queueing place with service rate 1.5; the think stage is an infinite-server queueing place with rate 0.5 (mean think time 2). Two immediate transitions move one token per firing between the two places, so the token flow reproduces a machine-repairman (finite-population) model whose exact solution is available from `SolverMVA` on the equivalent `Delay+Queue` network:

```
N <- 4
model <- Network$new('QueueingPetriNet')
# Block 1: two queueing places (set_service installs the embedded queue)
cpu   <- Place$new(model, 'CPU', SchedStrategy$FCFS)     # single-server queue
think <- Place$new(model, 'Think', SchedStrategy$INF)    # infinite-server queue
jobs  <- ClosedClass$new(model, 'Jobs', N, think, 0)
cpu$set_service(jobs, Exp(1.5))      # embedded FCFS service, rate 1.5
think$set_service(jobs, Exp(0.5))    # embedded think time, mean 2
# Block 2: two immediate transitions cycling one token per firing
toCPU <- Transition$new(model, 'toCPU')
m1 <- toCPU$add_mode('m1')
toCPU$set_timing_strategy(m1, TimingStrategy$IMMEDIATE)
toCPU$set_enabling_conditions(m1, jobs, think, 1)
toCPU$set_firing_outcome(m1, jobs, cpu, 1)
toThink <- Transition$new(model, 'toThink')
m2 <- toThink$add_mode('m2')
toThink$set_timing_strategy(m2, TimingStrategy$IMMEDIATE)
toThink$set_enabling_conditions(m2, jobs, cpu, 1)
toThink$set_firing_outcome(m2, jobs, think, 1)
# Block 3: topology and initial marking
P <- model$init_routing_matrix()
P$set(jobs, think, toCPU, 1.0)
P$set(jobs, toCPU, cpu, 1.0)
P$set(jobs, cpu, toThink, 1.0)
P$set(jobs, toThink, think, 1.0)
model$link(P)
think$set_initial_state(N)   # all tokens start in the think place
cpu$set_initial_state(0)
# Block 4: solution. Queueing places are simulated by LDES
print(LDES$new(model, seed = 23000, samples = 200000)$avg_table())
# Exact cross-check on the equivalent Delay + M/M/1 network
ref   <- Network$new('ref')
queue <- Queue$new(ref, 'CPU', SchedStrategy$FCFS)
delay <- Delay$new(ref, 'Think')
cc    <- ClosedClass$new(ref, 'Jobs', N, delay, 0)
```

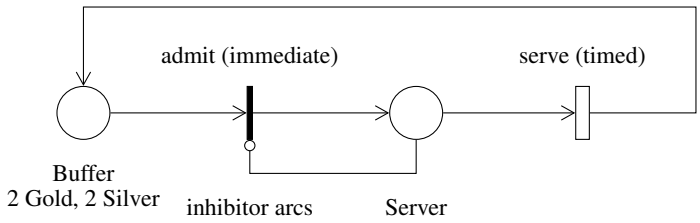


Figure 2.19: The colored GSPN of Tutorial 17. The solid thin bar is the immediate transition, the hollow bar the timed one, and the arc ending in a small circle is the inhibitor pair that keeps the server to one token: each of `admit`'s two modes is inhibited by a token of *either* color in `Server`.

```
delay$set_service(cc, Exp(0.5))
queue$set_service(cc, Exp(1.5))
ref$link(serial_routing(delay, queue))
print(MVA$new(ref)$avg_table())
```

The initial marking is `set_initial_state()` on a `Place`. A net written elsewhere in PNML is reachable too, without R having to understand it:

```
capi_run(c('-f', 'net.pnml', '-i', 'pnml', '-s', 'ctmc', '-a', 'avg', '-o', 'json'))
```

The token counts reported for the two queueing places match the exact finite-population result to within simulation error: The metric reported for a queueing place is its *token count*, that is the total marking summed over the tokens waiting, in service and in the depository; for the CPU place this equals the mean number in an M/M/1 queue and for the think place the mean number in service at the infinite-server stage. Queueing places are simulated by the `LDES` solver, which supports the `FCFS`, `LCFS`, `SIRO` and `INF` embedded scheduling strategies (single- or multiserver) with exponential, deterministic, immediate and renewal phase-type service. Because a queueing place embeds a service station inside a place, it cannot be represented as a single node in `JMT`; a queueing-place model is rejected by `SolverJMT` and must be expanded into a place plus an adjacent queueing station to be simulated there.

2.17 Tutorial 17: A colored generalized stochastic Petri net

A *generalized stochastic Petri net* (GSPN) admits two kinds of transition: *timed* ones, which fire after a random delay, and *immediate* ones, which fire as soon as they are enabled and consume no time at all [1]. It becomes *colored* when the tokens carry a type. In LINE a token color is simply a job class, so a place holds one marking per color and a transition declares one *mode* per color it serves, with its own firing law, enabling arcs and inhibitor arcs.

This tutorial builds the smallest net that uses all of it. Two colors, Gold and Silver, two tokens of each, circulate between a `Buffer` place and a `Server` place. The immediate transition `admit` has one mode per color and moves a waiting token into the server; both of its modes carry inhibitor arcs on *both* colors, so a token is admitted only into an empty server and the server becomes a mutual-exclusion resource. When both colors are waiting, which here is always, the conflict is resolved by the *firing weights*: Gold carries weight 2 against Silver's 1. The timed transition `serve` also has one mode per color and returns the token to the buffer at a color-dependent rate, $\mu_{\text{Gold}} = 3$ against $\mu_{\text{Silver}} = 1.5$:

```
model <- Network$new('ColoredGSPN')
# Block 1: declare every node before parameterizing any of it
buf   <- Place$new(model, 'Buffer')
srv   <- Place$new(model, 'Server')
admit <- Transition$new(model, 'admit')
serve <- Transition$new(model, 'serve')
# Block 2: two token colors, both starting in the buffer
gold  <- ClosedClass$new(model, 'Gold', 2, buf, 0)
```

```
silver <- ClosedClass$new(model, 'Silver', 2, buf, 0)
# Block 3: the immediate transition, one mode per color
for (col in list(list(gold, 2.0), list(silver, 1.0))) {
  m <- admit$add_mode(col[[1]]$name)
  admit$set_timing_strategy(m, TimingStrategy$IMMEDIATE)
  admit$set_enabling_conditions(m, col[[1]], buf, 1)
  admit$set_inhibiting_conditions(m, gold, srv, 1)
  admit$set_inhibiting_conditions(m, silver, srv, 1)
  admit$set_firing_outcome(m, col[[1]], srv, 1)
  admit$set_firing_weights(m, col[[2]])
}
# Block 4: the timed transition, one mode per color
for (col in list(list(gold, 3.0), list(silver, 1.5))) {
  m <- serve$add_mode(col[[1]]$name)
  serve$set_distribution(m, Exp(col[[2]]))
  serve$set_enabling_conditions(m, col[[1]], srv, 1)
  serve$set_firing_outcome(m, col[[1]], buf, 1)
}
# Block 5: topology and initial marking
P <- model$init_routing_matrix()
for (c in list(gold, silver)) {
  P$set(c, buf, admit, 1.0)
  P$set(c, admit, srv, 1.0)
  P$set(c, srv, serve, 1.0)
  P$set(c, serve, buf, 1.0)
}
model$link(P)
buf$set_initial_state(c(2, 2))
srv$set_initial_state(c(0, 0))
# Block 6: exact solution, and a simulation cross-check
print(CTMC$new(model, cutoff = 4)$avg_table())
print(SSA$new(model, seed = 23000, samples = 200000)$avg_table())
```

The immediate transition is removed before the chain is solved: stochastic complementation eliminates the vanishing markings, here 136 of them, and leaves 8 tangible states that `SolverCTMC` solves exactly. The simulation reproduces them: Every number here can be checked by hand, which is the point of building the net this way. Admission costs no time, so the server is never idle and $U_{\text{Gold}} + U_{\text{Silver}} = 1$. Both colors always have a token waiting, so every admission is a weighted draw and the completions split in the ratio of the firing weights, $X_{\text{Gold}} = 2X_{\text{Silver}}$. The mean service time is therefore $\frac{2}{3} \cdot \frac{1}{3} + \frac{1}{3} \cdot \frac{2}{3} = \frac{4}{9}$, so the two colors complete at $\frac{9}{4} = 2.25$ jobs per unit time in total, 1.5 Gold and 0.75 Silver, and $U_c = X_c/\mu_c = 0.5$ for each. What remains, 1.5 tokens of each color, is waiting in the buffer. Change one firing weight and the whole table moves with it: at equal weights the two colors complete at 1 each instead, and the buffer holds 1.667 Gold against 1.333 Silver.

Bibliography

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.

[2] F. Bause. Queueing Petri nets - a formalism for the combined qualitative and quantitative analysis of systems. In *Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM)*, pages 14–23, 1993.

[3] A. Bobbio, A. Horváth, M. Scarpa, and M Telek. Acyclic discrete phase type distributions: properties and a parameter estimation algorithm. *Perform. Eval.*, 54(1):1–32, 2003.

[4] F. P. Kelly. Loss networks. *Annals of Applied Probability*, 1(3):319–378, 1991.

[5] KT Marshall. Some relationships between the distributions of waiting time, idle time and interoutput time in the gi/g/1 queue. *SIAM Journal on Applied Mathematics*, 16(2):324–327, 1968.

[6] Keith W. Ross. *Multiservice Loss Models for Broadband Telecommunication Networks*. Springer, London, 1995.

Index

activity, 25
APH distribution, 12

blocking probability, 30
breakdowns, 27

Cache node, 18
caching, 17
chain, 17
class switching, 16
ClassSwitch node, 16
closed network, 13
ClosedClass class, 14
cluster, 29
Cluster class, 29
colored Petri net, 33
CTMC solver, 12
cumulative distribution function, 22

Delay node, 14
delayed hits, 20
departure process, 24
distributions
    APH, 12
    Erlang, 11
    Exp, 11
    Immediate, 18
    Prior, 27
    Replayer, 12
    Zipf, 17
DPS scheduling, 17
DROP, 30

entry, 25
ENV solver, 26
Environment class, 27
Erlang distribution, 11
Exp distribution, 11

FCFS scheduling, 10
finite capacity region, 30
firing mode, 33
firing weight, 33

FLD solver, 27
fluid approximation, 22

generalized stochastic Petri net, 33
GPS scheduling, 17

HOL scheduling, 17

Immediate distribution, 18
immediate feedback, 17
INF scheduling, 17
infinitesimal generator, 14
inhibitor arc, 33
insensitivity, 21

layered queueing network, 25
LayeredNetwork class, 25
LCFS scheduling, 17
LCFSPR scheduling, 21
LDES solver, 1, 10, 12, 30
LN solver, 25
load balancing, 14
loss network, 30

M/M/1 queue, 10
machine interference problem, 13
MAM solver, 13
Markov chain
    continuous-time, 13
Marshall formula, 25
model classes
    ClosedClass, 14
    Cluster, 29
    Environment, 27
    LayeredNetwork, 25
    OpenClass, 11
model features
    breakdowns, 27
    caching, 17
    class switching, 16
    delayed hits, 20
    finite capacity region, 30
    immediate feedback, 17

load balancing, 14
multiclass models, 11
multiserver stations, 29
parameter uncertainty, 27
random environment, 26
synchronous calls, 25
trace-driven service, 12
warm start, 1, 7
multiclass models, 11
multiserver stations, 29

NC solver, 16
next reaction method, 30
node types
Cache, 18
ClassSwitch, 16
Delay, 14
Place, 32
Queue, 11
Router, 14
Sink, 11
Source, 11
Transition, 32, 33
normalizing constant, 30

OpenClass class, 11
optimization, 23
options
cutoff, 13
seed, 11
verbose, 12

parameter uncertainty, 27
percentiles, 21
performance metrics
Queue length, 11
Residence time, 11
Response time, 11
Throughput, 11
Utilization, 11
Place node, 32
Poisson process, 24
posterior analysis, 27
Prior distribution, 27
PROB routing, 23
product form, 30
PS scheduling, 14

Queue length (QLen), 11
Queue node, 11
queueing Petri net, 31
queueing place, 31

random environment, 26
reentrant line, 16
reference station, 14
reference task, 25
replacement policy
LRU, 17
Replayer distribution, 12
Residence time (ResidT), 11
Response time (RespT), 11
response time distribution, 22
retrieval latency, 21
retrieval system, 20
Router node, 14
routing strategies
PROB, 23
RROBIN, 15
RROBIN routing, 15

sample path, 24
scheduling strategies
DPS, 17
FCFS, 10
GPS, 17
HOL, 17
INF, 17
LCFS, 17
LCFSPR, 21
PS, 14
SIRO, 17
service time, 10
Sink node, 11
SIRO scheduling, 17
solution methods
fluid approximation, 22
Marshall formula, 25
next reaction method, 30
TTL algorithm, 19
solvers
CTMC, 12
ENV, 26
FLD, 27
LDES, 1, 10, 12, 30
LN, 25
MAM, 13
NC, 16
SSA, 19
UQ, 27
Source node, 11

squared coefficient of variation, 12
SSA solver, 19
state space
    generation, 14
    truncation, 12
stateful node, 24
stochastic Petri net, 31
synchronous calls, 25

Throughput (Tput), 11
trace-driven service, 12
Transition node, 32, 33
TTL algorithm, 19

UQ solver, 27
Utilization (Util), 11

warm start, 1, 7

Zipf distribution, 17