LINE: Queueing Analysis Algorithms

A Primer – C++ Edition

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

Contents

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.

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.

1.1.1 Obtaining and installing LINE

The C++ edition is compiled from source with CMake. From the root of the source tree:

```
cmake -S cpp -B cpp/build -DCMAKE_BUILD_TYPE=Release
cmake --build cpp/build -j4
```

The library interface is header-only: a program adds `cpp/include` to its include path and needs no separate LINE library to link against. The build also produces the `line-cli` front end, which solves a model file from the command line, and the `line-examples` example runner.
The latest releases and the source code are available from http://line-solver.sf.net.

1.1.2 Software requirements

The C++ edition requires a C++17 compiler (GCC 9 or later, or Clang 10 or later), CMake 3.16 or later, and the Boost.Multiprecision headers (1.71 or later), which are header-only and supply the exact (`cpp_rational`) and high-precision (`cpp_bin_float`) arithmetics behind the multiprecision API. Two numerical libraries are located at configure time and, although both are nominally optional, in practice both are wanted. LAPACK (`-DLINE_MP_USE_LAPACK`, on by default) supplies the eigenvalues, singular values and Schur decompositions; without it those entry points, and everything reached through them – the fluid second-moment methods above all – refuse by name at run time. HiGHS (`-DLINE_MP_USE_HIGHS`, on by default) supplies the sparse simplex and the convex quadratic program behind the QRF bounds and the NLP entry points; without it the dense fallback tableau caps those bounds at a few hundred columns. GMP and MPFR replace the Boost backends

when -DLINE_MP_USE_GMP=ON is given: they are faster, but LGPL, so they are off by default and the default build carries no LGPL obligation. On Debian and Ubuntu the whole set is installed with

```
sudo apt install build-essential cmake libboost-dev liblapack-dev libblas-dev
sudo apt install libhighs-dev    # Debian and Ubuntu 25.04 or later
```

the equivalents being gcc-c++ make cmake boost-devel lapack-devel blas-devel with dnf and cmake boost lapack highs with Homebrew. Older Ubuntu releases have no HiGHS package and build it from source (https://github.com/ERGO-Code/HiGHS), as the user manual details. CMake reports each dependency it found on its own line_mp: line at configure time; those lines are the record of what the resulting build can do.
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. Place JMT.jar in common/, or point the LINE_JMT_DIR environment variable at the folder holding it; this edition does not download it.
- *LQNS* (https://github.com/layeredqueuing/V6), version 6.2.28 or later, is required by the LQNS and QNS wrappers for layered models. 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. This edition has no computer algebra system of its own, so the backend is the only way to run symbolic analysis from C++.

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 and JMT, the LQNS binaries, and the SageMath symbolic backend) are reachable. Run it once after installing:

```
cpp/build/line-cli --install
```

On a machine where every dependency is reachable, the check reports:

```
Checking LINE (C++)...
  line-cli 0.1.0
Checking Java runtime (JMT wrapper)...
  /usr/bin/java
Checking JMT...
  common/JMT.jar
Checking LQNS...
  Layered Queueing Network Analyser, Version 6.2.31
Checking QNS (qnsolver)...
Checking symbolic backend (line-sage-rest)...
Success. LINE is ready to use.
```

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: the C++ edition is header-only and its native solvers stand alone, so every dependency above backs one optional wrapper or backend. 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 and QNS wrappers require the corresponding commands on PATH.

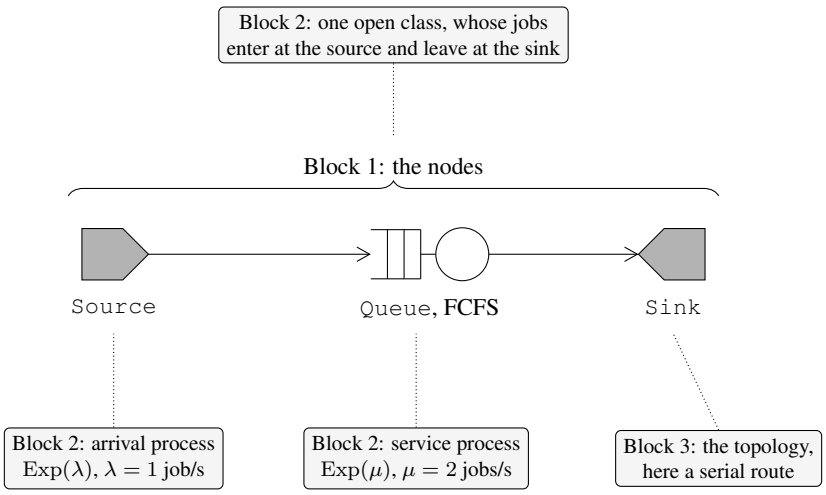

Figure 1.1: The M/M/1 queue, annotated with the blocks every LINE model is built from. A Poisson stream of rate λ feeds one FCFS station with exponential service of rate μ, and completed jobs leave through the sink. The fourth block, the solution, is the choice of solver and has no counterpart in the 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. Every script in Chapter 2 is laid out in these four blocks and marks them in its 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, and LINE returns exactly those values. 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 a `samples` budget, which trades accuracy against time. Verbosity is controlled globally through `GlobalConstants`.

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). Non-exponential timing is handled by phase-type expansion or by fitting: `aph_fit_mean_scv(m,scv)` 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

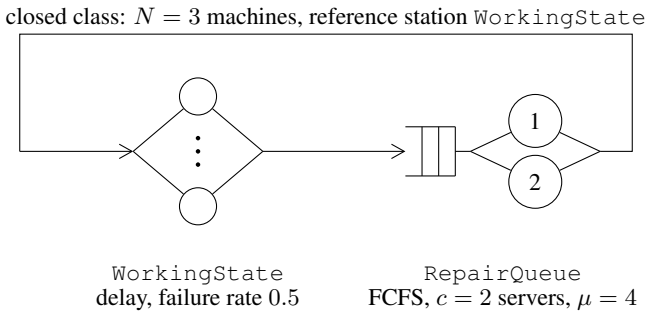

Figure 1.2: A closed network: the machine-repairman model. A fixed population of $N = 3$ machines alternates between a working state, modelled as a delay, and a two-server FCFS repair station. No job enters or leaves, so exactly N jobs are present at all times.

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 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.

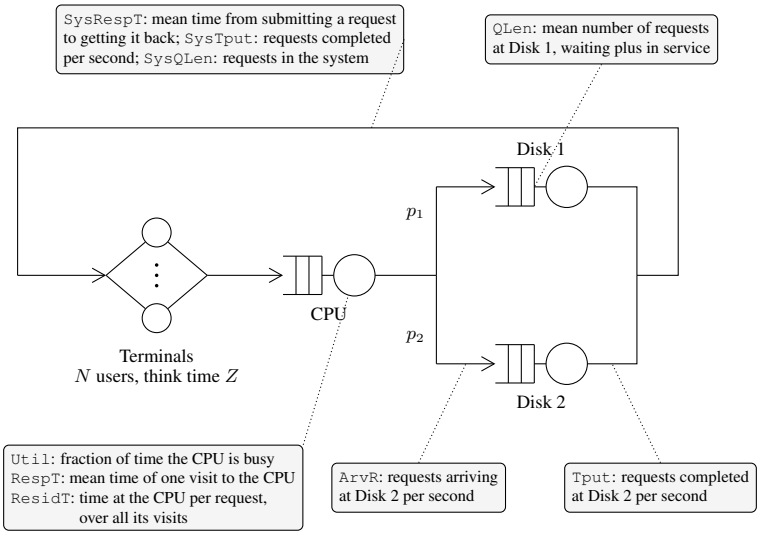

Figure 1.3: The metrics LINE reports, named on a closed network in which N users at terminals submit requests that are served by a CPU and by one of two disks before returning to their user. `QLen`, `Util`, `RespT`, `ResidT`, `ArvR` and `Tput` are reported per station and class; the system metrics are 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. *Meta-solvers* 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
Meta-solvers		
`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`	layered models
`QNS`	`qnsolver` (LQNS suite)	closed product-form models, multiserver approximations

A meta-solver 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

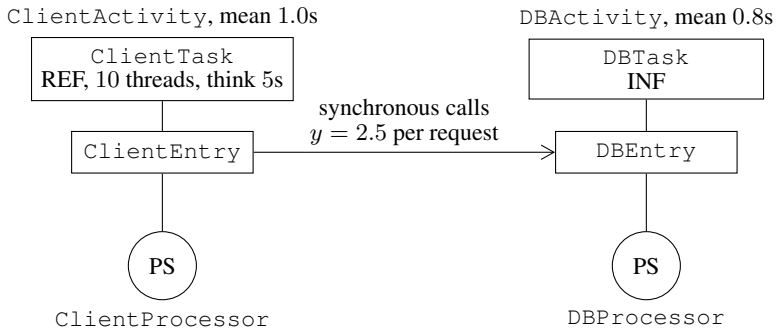

Figure 1.4: A layered queueing network: a two-tier client–server application. Each task offers one entry, runs one activity, and is hosted by its own processor; the client activity issues on average 2.5 synchronous calls per request to the database entry.

`LQNS` and `QNS` require 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.

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).

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` meta-solver takes a factory producing an inner solver for each stage and returns metrics averaged over the stationary distribution of the environment.

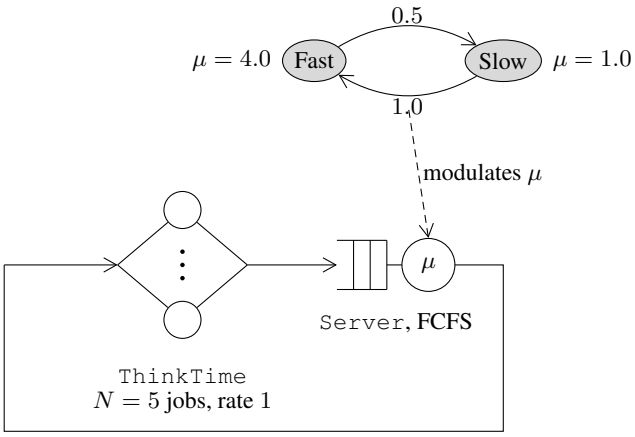

Figure 1.5: A model in a random environment: a two-state Markov chain alternates between a fast and a slow mode, each a complete stage network, and modulates the service rate of the queue. The environment is fast $2/3$ of the time and slow $1/3$ of the time.

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 compiled `line-cli` front end solves models from the native JSON format and reports results as text or JSON. For example, mean-value analysis is run as follows:

```
cpp/build/line-cli -f example.json -s mva -a avg
```

The commands `cpp/build/line-cli -help` and `cpp/build/line-cli -list-api` are the authoritative inventories of the front-end options and the API functions available in the current build. 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 sixteen 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 (`LINE-user-cpp.pdf`), which expands every topic in this primer in greater depth, and to the LINE example manual (`LINE-examples-cpp.pdf`), which catalogues the runnable example suite.

Chapter 2

Tutorials

This chapter walks through sixteen complete models, from a single M/M/1 queue to layered, cached, and Petri-net models. Each tutorial is self-contained: it states the model, gives the full code, and comments the resulting metrics. 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 (`LINE-internals-cpp.pdf`).

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 JMT-based 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

```
Network model("M/M/1");
// Block 1: nodes
Source source(model, "Source");
Queue queue(model, "Queue", SchedStrategy::FCFS);
Sink sink(model, "Sink");
// Block 2: classes
OpenClass jobclass(model, "Class1");
source.set_arrival(jobclass, Exp(1.0));
queue.set_service(jobclass, Exp(2.0));
// Block 3: topology
Routing P;
P.set(source, queue, 1.0);
P.set(queue,  sink,  1.0);
model.link(P);
// Block 4: solution (the C++ port has no JMT bridge; SSA is its simulator)
ssa::SsaOptions opt;
opt.samples = 10000;
opt.seed = 23000;
const ssa::SsaSolution res = ssa::solver_ssa(model.get_struct(), opt);
```

In the example, `source` and `sink` are arrival and departure points of jobs; `queue` is a queueing

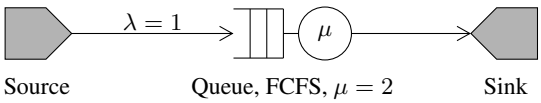

Figure 2.1: The M/M/1 queue of Tutorial 1: a Poisson source feeds a single FCFS station with exponential service, and completed jobs leave through the sink.

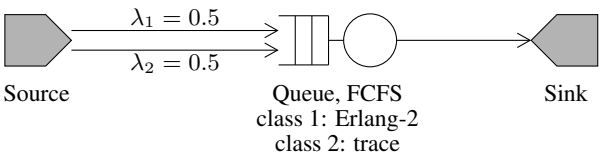

Figure 2.2: The multiclass M/G/1 queue of Tutorial 2: two open classes share one FCFS station, class 1 with Erlang-2 service and class 2 with a trace-driven service time.

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 rate parameter $\lambda = 2.0$; finally, the `solver_ssa` call solves for average performance measures with the native simulator, 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`) The result is the matrices `QN`, `UN`, `RN` and `TN` rather than a printed table; the front end formats them as `line-cli -f model.json -s ssa -a avg`. A pre-defined gallery of classic models is also available, for example There is no model gallery in the C++ port; the model above is written out in full. returns a M/M/1 queue with 50% utilization.

If we want to select a particular row of the `AvgResult` data structure, we can use direct object-based indexing or a filtering method. The results are matrices indexed by (station, class), so a single figure is one indexed read and there is no row filter:

```
res.QN(model.station_index(queue) - 1, jobclass - 1);
res.RN(model.station_index(queue) - 1, jobclass - 1);
```

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 non-exponential 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 $2s$.

To run this example, let us first change the working directory to the `examples/` folder. Then we specify the node block

```
Network model("M/G/1");
Source source(model, "Source");
Queue queue(model, "Queue", SchedStrategy::FCFS);
Sink sink(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

```
OpenClass jobclass1(model, "Class1");
OpenClass jobclass2(model, "Class2");

source.set_arrival(jobclass1, Exp(0.5));
source.set_arrival(jobclass2, Exp(0.5));

queue.set_service(jobclass1, Distrib<double>::erlang_fit(1.0, 1.0 / 3.0));
queue.set_service(jobclass2, Replayer(samples));
```

`replayer` takes the trace as a vector of samples: the C++ port does not read a trace file. 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

```
Routing P;
for (std::size_t r : {jobclass1, jobclass2}) {
    P.set(r, r, source, queue, 1.0);
    P.set(r, r, queue,  sink,  1.0);
}
model.link(P);
```

and solve the model with JMT The simulation answer comes from `ssa::solver_ssa`: the C++ port has no JMT bridge. 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 [2] based on the trace The same figures are read off `AvgResult::QN`, `UN`, `RN` and `TN`. 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, checking with the `verbose` option the size of the resulting state space

```
mva::MvaOptions opt;
const mva::AvgResult<double> res =
    mva::solver_mva_run_analyzer(model.get_struct(), opt, Matrix<double>());
```

However, we see from the comparison with JMT that the errors of `CTMC` are rather large. Since the truncated state space consists of just 46 states, we can further increase the cutoff to 4, trading a slower solution time for higher precision

```
nc::NcSolverOptions opt;
const mva::AvgResult<double> res = nc::solver_nc_run_analyzer(model.get_struct(), opt);
```

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 exactly analyze open systems with an infinite state space, calling

```
mam::MamOptions opt;
const mva::AvgResult<double> res = mam::solver_mam_run_analyzer(model.get_struct(), opt);
```

The C++ matrix-analytic solver is a port of the same algorithms and calls no external library.

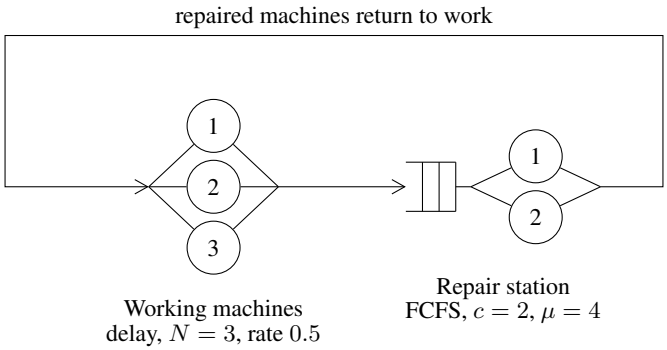

Figure 2.3: The machine interference model of Tutorial 3: $N = 3$ machines alternate between a working state, modelled as a delay, and a two-server FCFS repair station.

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:

```
const double S = 2, N = 3;
Network model("MRP");
// Block 1: nodes
Delay delay(model, "WorkingState");
Queue queue(model, "RepairQueue", SchedStrategy::FCFS);
queue.set_number_of_servers(S);
// Block 2: classes
ClosedClass cclass(model, "Machines", N, delay);
delay.set_service(cclass, Exp(0.5));
queue.set_service(cclass, Exp(4.0));
// Block 3: topology
Routing P;
P.set(delay, queue, 1.0);
P.set(queue, delay, 1.0);
model.link(P);
// Block 4: solution
ctmc::CtmcOptions opt;
const mva::AvgResult<double> res = ctmc::solver_ctmc_run_analyzer(model.get_struct(), opt);
```

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

```
WorkingState  Machines   2.6648   2.6648        2        2  1.3324  1.3324
RepairQueue   Machines  0.33516  0.16655  0.25154  0.25154  1.3324  1.3324
```

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

```
const ctmc::CtmcStateSpace<double> space =
    ctmc::ctmc_get_state_space(model.get_struct(), opt);
const ctmc::CtmcGenerator<double> gen =
    ctmc::ctmc_get_generator(model.get_struct(), opt);
```

which produces in output the state space of the model and the infinitesimal generator of the CTMC

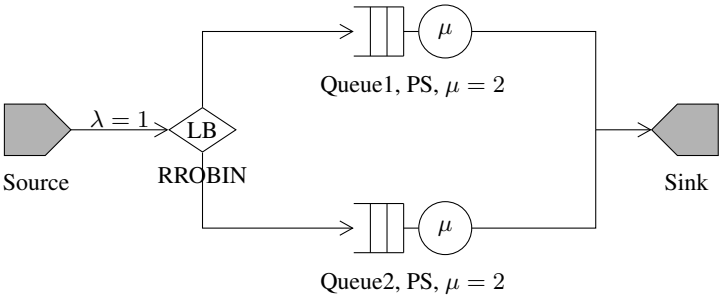

Figure 2.4: The load-balancing model of Tutorial 4: a router dispatches the open stream to two identical processor-sharing queues, first at random and then in round-robin order.

`space` carries the same state list and `gen.Q` the same generator; the front end prints them with `line-cli -s ctmc -a states` and `-a gen`. 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 `ctmc_get_infgen`, 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 `ctmc_get_state_space` associated to the source and destination states.

On this and larger infinite generators, we may also list individual non-zero transitions as follows The C++ port has no generator pretty-printer: the non-zero entries of `gen.Q` are read against the rows of the state space. The above printout helps in matching the state transitions to their rates.

To avoid having to inspect the `ctmc_get_state_space` variable to determine to which station a particular column refers to, we can alternatively use the more general invocation The per-node state spaces are the columns of the global one; `qn::from_marginal` enumerates the states of a single station. 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:

```
Network model("RRLB");
Source source(model, "Source");
Router lb(model, "LB");
Queue queue1(model, "Queue1", SchedStrategy::PS);
Queue queue2(model, "Queue2", SchedStrategy::PS);
Sink sink(model, "Sink");
```

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

```
OpenClass jobclass(model, "Class1");
```

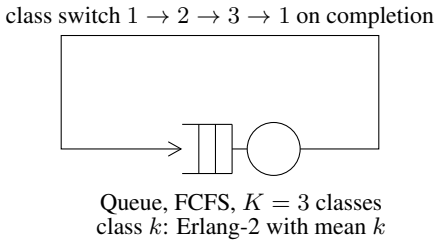

Figure 2.5: The re-entrant line of Tutorial 5: a single FCFS station is visited three times, the class switching on each completion so that every visit draws its service time from a different distribution.

```
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 non-probabilistic 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):

```
Routing P;
P.set(source, lb,     1.0);
P.set(lb,     queue1, 0.5);
P.set(lb,     queue2, 0.5);
P.set(queue1, sink,   1.0);
P.set(queue2, sink,   1.0);
model.link(P);
```

There is no `addLink` in C++: the arcs and their probabilities are the routing matrix, so an even split is written as two entries of 0.5. 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.

```
model.set_routing(lb, jobclass, RoutingStrategy::RROBIN);
```

The strategy replaces the probabilities on the router's outgoing arcs; there is no model reset, since `get_struct` re-derives everything on the next call. Lastly, we run again JMT and find that round-robin produces a visible decrease in response times, which are now around $0.56s$. The round-robin answer is again read off the `AvgResult` of whichever solver is run.

2.5 Tutorial 5: Modelling a re-entrant line

Let us now consider a simple example inspired to the classic problem of modeling *re-entrant lines*. This arises in manufacturing systems where parts (i.e., jobs) re-enter 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 re-enters 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.

```
Network model("RL");
Queue queue(model, "Queue", SchedStrategy::FCFS);

const std::size_t K = 3;
const double N[3] = {1, 0, 0};
std::vector<std::size_t> jobclass(K);
for (std::size_t k = 0; k < K; ++k) {
    jobclass[k] = model.add_closed_class("Class" + std::to_string(k + 1), N[k], queue);
    queue.set_service(jobclass[k],
                      Erlang(2.0 / double(k + 1), 2));
}

Routing P;
P.set(jobclass[0], jobclass[1], queue, queue, 1.0);
P.set(jobclass[1], jobclass[2], queue, queue, 1.0);
P.set(jobclass[2], jobclass[0], queue, queue, 1.0);
model.link(P);
```

`erlang` takes the phase rate and the number of phases, so an Erlang of mean k with two phases has phase rate $2/k$.
The class-switching rule is enforced automatically, LINE introducing a `ClassSwitch` node in the network when the model is linked.
We can now simulate the performance indexes for the different classes, for example using LINE's normalizing constant solver (`NC`)

```
nc::NcSolverOptions opt;
const mva::AvgResult<double> res = nc::solver_nc_run_analyzer(model.get_struct(), opt);
```

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 `AvgResult::CN` method, i.e., The chain-level answer is `res.CN` and `res.XN`, the system response time and throughput per class, aggregated over the chain by the solver. 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.
The composition of a chain is `sn.chains`, a per-chain membership row over the 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

```
model.raw_struct().classes[jobclass[0] - 1].completes = false;
model.raw_struct().classes[jobclass[1] - 1].completes = false;
```

This modification then gives the correct chain throughput, matching the one of `Class3` alone With the two classes marked as non-completing, the cycle is counted once and the system response time is the sum over the three visits.
In the model above, a job that re-enters 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 `queue.setImmediateFeedback(true)` (or passing a specific class) instead makes such class-switching self-loops *immediate*: the job retains the server and starts its next service without re-queueing, its throughput still being counted and a multi-phase 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.

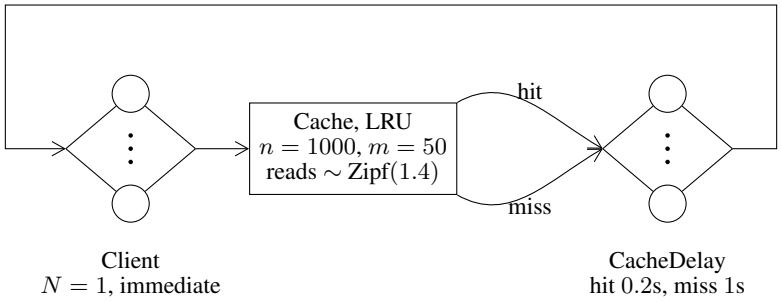

Figure 2.6: The cache-augmented network of Tutorial 6: a single client reads items from an LRU cache, and the read is charged a hit or a miss latency at the cache delay before the client is released.

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.

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:

```
Network model("QNC");
// Block 1: nodes
Delay clientDelay(model, "Client");
qn::CacheParam<double> par;
par.nitems = 1000;
par.itemcap.assign(1, 50);
par.replacestrat = ReplacementStrategy::LRU;
const std::size_t cacheNode = model.add_cache("Cache", par);
Delay cacheDelay(model, "CacheDelay");
```

The cache parameters are filled in before the node is created, so the popularity and the hit and miss classes of the next listings belong to the same `CacheParam`.

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:

```
// Block 2: classes
ClosedClass clientClass(model, "ClientClass", 1, clientDelay, 0);
ClosedClass hitClass(model, "HitClass", 0, clientDelay, 0);
ClosedClass missClass(model, "MissClass", 0, clientDelay, 0);
```

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

```
clientDelay.set_service(clientClass, Immediate());
cacheDelay.set_service(hitClass,  Exp(0.2));
cacheDelay.set_service(missClass, Exp(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

```
// Zipf(1.4) popularity over the 1000 items, as a probability vector
std::vector<double> pop(par.nitems);
double norm = 0.0;
for (std::size_t i = 0; i < par.nitems; ++i) {
    pop[i] = 1.0 / std::pow(double(i + 1), 1.4);
    norm += pop[i];
}
for (std::size_t i = 0; i < par.nitems; ++i) pop[i] /= norm;
par.pread.assign(nclasses, std::vector<double>());
par.pread[clientClass - 1] = pop;
```

There is no `Zipf` object in the C++ port: the read probabilities are given directly. Finally, we ask that the job should become of class after a cache hit, and should become of class after a cache miss:

```
par.hitclass.assign(nclasses, 0);
par.missclass.assign(nclasses, 0);
par.hitclass[clientClass - 1]  = hitClass;
par.missclass[clientClass - 1] = missClass;
```

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

```
// Block 3: topology
Routing P;
P.set(clientClass, clientClass, clientDelay, cacheNode, 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(hitClass,  hitClass,  cacheNode, cacheDelay, 1.0);
P.set(missClass, missClass, cacheNode, cacheDelay, 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(hitClass,  clientClass, cacheDelay, clientDelay, 1.0);
P.set(missClass, clientClass, cacheDelay, clientDelay, 1.0);
```

The above routing strategy is finally applied to the model

```
model.link(P);
```

Solution block To solve the model, since JMT does not support cache modeling, we use the native simulation engine provided within LINE, the `SSA` solver:

```
// Block 4: solution
ssa::SsaOptions opt;
opt.samples = 20000;
opt.seed = 1;
const ssa::SsaSolution res = ssa::solver_ssa(model.get_struct(), opt);
```

The above script produces the following result The C++ simulator reports the same metrics in `res.QN`, `res.UN`, `res.RN` and `res.TN`. The departing flows from the are the miss and hit rates. Thus, the hit rate is 2.4554 jobs per unit time, while the miss rate is 0.50892 jobs per unit time.

Let us now suppose that we wish to verify the result with a longer simulation, for example with 10 times more samples. To this aim, we can use the automatic parallelization of `SSA`

```
ssa::SsaOptions opt;
```

```
opt.method = "para";     // the parallel engine, replications run concurrently
opt.samples = 20000;
opt.seed = 1;
const ssa::SsaSolution res = ssa::solver_ssa(model.get_struct(), opt);
```

This gives us a rather similar result, when run on a dual-core machine The parallel engine reports the same quantities; only the sampling error differs. The execution time is longer than usual at the first invocation of the parallel solver due to the time needed by MATLAB to bootstrap the parallel pool, in this example around 22 seconds. Successive invocations of parallel SSA normally take much less, with this example around 7 seconds each.

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`:

```
mva::MvaOptions opt;
const mva::CacheResult<double> res =
    mva::solver_mva_cache_analyzer(model.get_struct(), opt);
// res.hitprob[r] and res.missprob[r], per read class
```

Per-item hit probabilities are not tabulated by the C++ port; the per-class hit and miss probabilities are. The cache table reports an overall hit probability of about 0.771 for the requesting class

```
 Node     JobClass    List  ListCap  Items  HitProb  MissProb
 Cache  ClientClass      0       50   1000  0.77121   0.22879
```

The item table contains one row per (item, list); 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`).

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 back-end 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; `setRetrievalSystem` takes only the retrieval queue(s) and

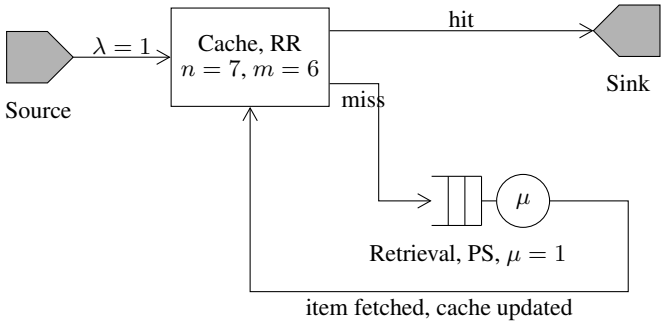

Figure 2.7: The delayed-hit cache of Tutorial 7: a miss is routed to a processor-sharing retrieval station, and only when the fetch completes is the cache updated and the request released.

installs the cache→queue→cache routing automatically. The retrieval service time is taken from the read class's service distribution at each queue, set with `set_service`.

```
const std::vector<double> accessProb{49, 49, 49, 49, 7, 1, 1};   // popularities
const std::size_t n = accessProb.size();

Network model("Retrieval");
Source source(model, "Source");
qn::CacheParam<double> par;
par.nitems = n;
par.itemcap.assign(1, 6);
par.replacestrat = ReplacementStrategy::RR;
const std::size_t cacheNode = model.add_cache("Cache", par);
Queue queue(model, "Queue", SchedStrategy::PS);
Sink sink(model, "Sink");

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

source.set_arrival(jobClass, Exp(1.0));
queue.set_service(jobClass, Exp(1.0));
model.set_retrieval_system(cacheNode, jobClass, missClass,
                           std::vector<std::size_t>{queue});

Routing P;
P.set(jobClass,  jobClass,  source,    cacheNode, 1.0);
P.set(hitClass,  hitClass,  cacheNode, sink,      1.0);
P.set(missClass, missClass, cacheNode, sink,      1.0);
model.link(P);
```

The read probabilities and the hit and miss classes are set on `par` before the node is created; `set_retrieval_system` then creates one retrieval class per item, each inheriting the read class's service at the retrieval station.

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::NcSolverOptions opt;
const mva::AvgResult<double> res = nc::solver_nc_run_analyzer(model.get_struct(), opt);
```

The hit and miss probabilities are those of the cache analyzer, `mva::CacheResult`; the delayed share is one minus their sum.

Approximate analysis and latency `MVA` uses the fixed-point heuristic to obtain the expected retrieval latency Z, the mean delay incurred by misses and delayed hits. This latency is reported in the `ResidT`

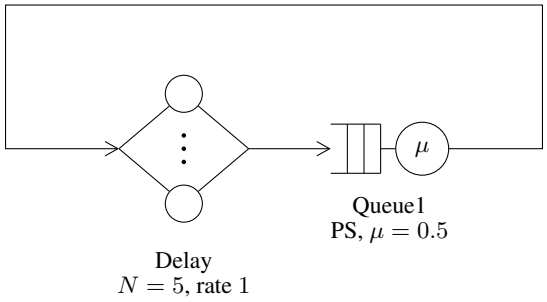

Figure 2.8: The closed network of Tutorial 8, used to compare the response time distribution of the processor-sharing queue obtained by fluid analysis and by simulation.

column of `getAvgCacheTable` and is populated by the `MVA`, `NC`, `CTMC`, `SSA` and `LDES` solvers: the analytical solvers obtain it from the retrieval recurrence, the simulators measure it directly on the sample path. Solvers that do not model the retrieval sub-network leave the column unset (`NaN`).

```
mva::MvaOptions opt;
const mva::CacheResult<double> res =
    mva::solver_mva_cache_analyzer(model.get_struct(), opt);
```

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.

```
Network model("Model");
// Block 1: nodes
Delay delay(model, "Delay");
Queue queue(model, "Queue1", SchedStrategy::PS);
```

There is a single class consisting of 5 jobs that circulate between the two stations, taking exponential service times at both.

```
ClosedClass jobclass(model, "Class1", 5, delay, 0);
delay.set_service(jobclass, Exp(1.0));
queue.set_service(jobclass, Exp(0.5));

Routing P;
P.set(delay, queue, 1.0);
P.set(queue, delay, 1.0);
model.link(P);
```

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 JMT. To do so, we call the `solver_ctmc_cdf_respt` method The response-time distribution is a front-end analysis in the C++ port, `line-cli -f model.json -s fluid -a cdf`, also available for `-s ctmc` and `-s nc`. In-process, `ctmc::solver_ctmc_cdf_respt` returns one curve per (station, class). .

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

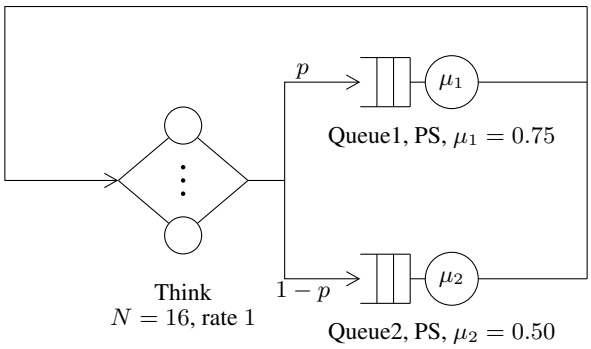

Figure 2.10: The load-balancing closed network of Tutorial 9: the routing probability p that splits the think-time output between the two queues is the decision variable of the optimization.

The 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.
The curves are sequences of (probability, time) points; plotting them is left to the caller. which produces the graph shown in Figure 2.9. The graph shows that, although the simulation refers to a transient, while the fluid approximation refers to steady-state, there is a tight matching between the two response time distributions.

We can readily compute the percentiles from the data structures, e.g., for the 95th and 99th percentiles of the simulated distribution A percentile is read off the same curve: the largest time whose cumulative probability is below the level. That is, 95% of the response times at the PS queue (node 2, class 1) are less than or equal to 27.0222 time units, while 99% are less than or equal to 41.8743 time units.

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.
The search below is written directly in the host program for this scalar tutorial; reusable searches are available through the C++ `line::opt` API. Within the function definition, we instantiate the two queues and the delay station

```
Network model("LoadBalCQN");
// Block 1: nodes
Delay delay(model, "Think");
Queue queue1(model, "Queue1", SchedStrategy::PS);
Queue queue2(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:

```
// Block 2: classes
ClosedClass cclass(model, "Job1", 16, delay);
delay.set_service(cclass, Exp(1.0));
queue1.set_service(cclass, Exp(0.75));
queue2.set_service(cclass, Exp(0.50));
```

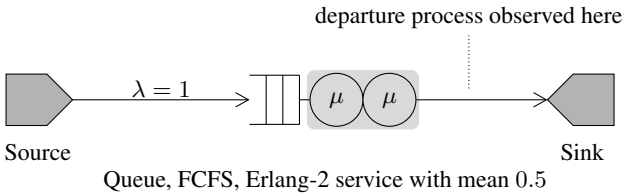

Figure 2.11: The $M/E_2/1$ queue of Tutorial 10: the two shaded circles are the phases of the Erlang-2 service time, and the departure process is observed at the station output.

We 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:

```
// Block 3: topology
Routing P;
P.set(queue1, delay, 1.0);
P.set(queue2, delay, 1.0);
```

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:

```
// Block 4: solution -- the objective, as a function of the split p
const auto objFun = [&](double p) {
    Routing Pp = P;
    Pp.set(delay, queue1, p);
    Pp.set(delay, queue2, 1.0 - p);
    model.link(Pp);
    mva::MvaOptions opt;
    opt.method = "exact";
    const mva::AvgResult<double> r =
        mva::solver_mva_run_analyzer(model.get_struct(), opt, Matrix<double>());
    return r.CN[cclass - 1];        // system response time
};
```

`link` may be called again with a new matrix, which is what re-linking amounts to here. Lastly, we optimize the function we defined

```
// golden-section search on [0, 1]
double a = 0.0, b = 1.0;
const double phi = 0.5 * (std::sqrt(5.0) - 1.0);
double c = b - phi * (b - a), d = a + phi * (b - a);
double fc = objFun(c), fd = objFun(d);
while (b - a > 1e-8) {
    if (fc < fd) { b = d; d = c; fd = fc; c = b - phi * (b - a); fc = objFun(c); }
    else         { a = c; c = d; fc = fd; d = a + phi * (b - a); fd = objFun(d); }
}
const double p_opt = 0.5 * (a + b);
```

This scalar tutorial writes out the golden-section search directly; reusable model-level searches are also available through the C++ `line::opt` API.

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 inter-departure 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. There is no

model gallery in the C++ port; the M/Er/1 of this tutorial is built with `add_source`, `add_queue` and `add_sink` as in Tutorial 1, with an Erlang service. We now extract 50,000 samples from simulation based on the underpinning continuous-time Markov chain

```
ctmc::CtmcOptions opt;
opt.cutoff = 150;
const ctmc::CtmcSamplePath<double> sa =
    ctmc::solver_ctmc_sample_sys(model.get_struct(), opt, 100000, 23000);
```

The returned data structure supplies information about the stateful nodes (here `source` and `queue`) at each of the 50,000 instants of sampling, together with the events that have been collected at these instants. `sa` carries three parallel vectors: `t`, the time each state was entered, `state`, its index in the state space, and `event`, the synchronization that fired to leave it. As an example, the first two events occur both at timestamp 0 and indicate a departure event from node 1 (the type `EventType.DEP` maps to `event: DEP`) followed by an arrival event at node 2 (the type `EventType.ARV` maps to `event: ARV`) which accepts it always (`prob: 1`).
Each recorded event names the node it happened at, its type (an arrival or a departure), and the class involved, which is what the filter below selects on.

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

```
// keep the departures of the queue: sa.event[k] indexes the synchronization
// that fired, whose active half names the node and the event type
std::vector<double> depTimes;
for (std::size_t k = 0; k < sa.event.size(); ++k) {
    const qn::Sync<double>& sy = sa.chain.chain.sync[sa.event[k]];
    if (sy.active.node == ind && sy.active.event == EventType::DEP)
        depTimes.push_back(sa.t[k]);
}
```

Followed by a calculation of the time series of inter-departure times, whose leading entries are

```
std::vector<double> interDep(depTimes.size() - 1);
for (std::size_t k = 1; k < depTimes.size(); ++k)
    interDep[k - 1] = depTimes[k] - depTimes[k - 1];
```

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

```
double m = 0.0, m2 = 0.0;
for (double x : interDep) { m += x; m2 += x * x; }
m /= double(interDep.size());
m2 /= double(interDep.size());
const double scv = (m2 - m * m) / (m * m);
```

which evaluates to 0.8856. Using Marshall's exact formula for the $GI/G/1$ queue [4], 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:

```
lqn::LqnBuilder<double> lqn;
```

Next, we define the processors and tasks:

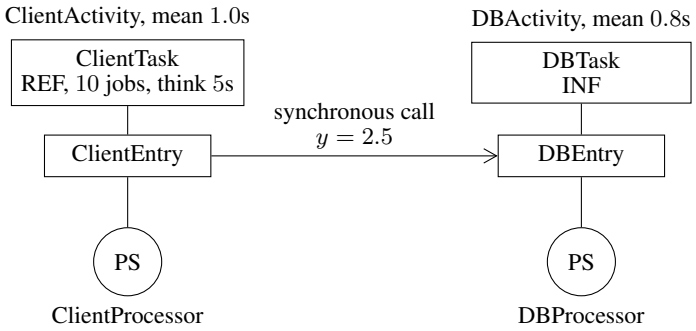

Figure 2.12: The layered network of Tutorial 11: the client task issues on average 2.5 synchronous calls per request to the database entry, and each task runs on its own processor.

```
// Processors
lqn.processor("ClientProcessor", 1, SchedStrategy::PS);
lqn.processor("DBProcessor", 1, SchedStrategy::PS);

// Tasks
lqn.task("ClientTask", 10, SchedStrategy::REF, "ClientProcessor");
lqn.think_time("ClientTask", Exp(5.0));   // 5-second think time
lqn.task("DBTask", std::numeric_limits<double>::infinity(),
         SchedStrategy::INF, "DBProcessor");
```

Now we define the entries that represent service interfaces:

```
lqn.entry("ClientEntry", "ClientTask");
lqn.entry("DBEntry", "DBTask");
```

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

```
// Client activity: processes the request and calls the database
lqn.activity("ClientActivity", Exp(1.0), "ClientTask");
lqn.bound_to("ClientActivity", "ClientEntry");
lqn.sync_call("ClientActivity", "DBEntry", 2.5);   // 2.5 DB calls on average

// Database activity: serves the request and replies
lqn.activity("DBActivity", Exp(0.8), "DBTask");
lqn.bound_to("DBActivity", "DBEntry");
lqn.replies_to("DBActivity", "DBEntry");
```

Now we solve the layered network using the LN solver with MVA applied to each layer:

```
ln::LnOptions opt;                  // layer_solver defaults to "mva"
ln::SolverLN<double> solver(lqn.build(), opt);
const ln::LnSolution<double> res = solver.get_ensemble_avg();
```

The solution carries the same per-element metrics as vectors indexed by processor, task, entry and activity: `res.QN`, `res.UN`, `res.RN`, `res.WN` and `res.TN`, with the companion `res.defined_Q`, `res.defined_U`, `res.defined_R`, `res.defined_W` and `res.defined_T` flags marking the entries that are meaningful for each element type.

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

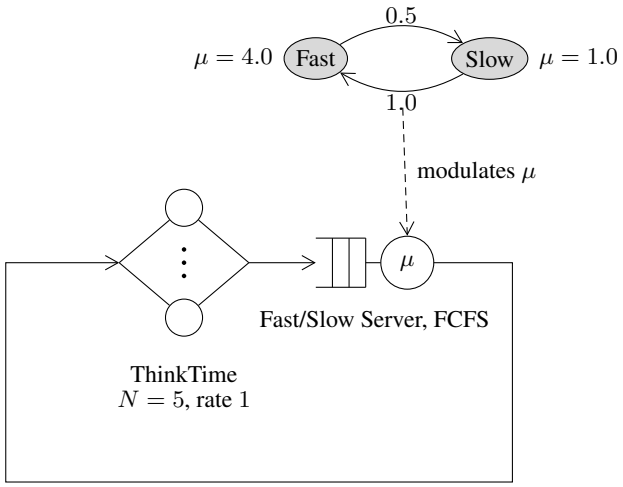

Figure 2.13: The random environment of Tutorial 12: a two-state Markov chain alternates between a fast and a slow mode, modulating the service rate of the queue in the underlying closed network.

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:

```
Network baseModel("BaseModel");
const std::size_t delay = baseModel.add_delay("ThinkTime");
const std::size_t queue = baseModel.add_queue("Fast/Slow Server", SchedStrategy::FCFS);
const double N = 5;
const std::size_t jobclass = baseModel.add_closed_class("Jobs", N, delay);
baseModel.set_service(delay, jobclass, Exp(1.0));
baseModel.set_service(queue, jobclass, Exp(2.0));
Routing P;
P.set(delay, queue, 1.0);
P.set(queue, delay, 1.0);
baseModel.link(P);
```

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:

```
env::Environment<double> environment("ServerModes", 2);

Network fastModel = baseModel;
fastModel.set_service(queue, jobclass, Exp(4.0));
environment.set_stage(0, "Fast", "operational", fastModel.get_struct());

Network slowModel = baseModel;
slowModel.set_service(queue, jobclass, Exp(1.0));
environment.set_stage(1, "Slow", "degraded", slowModel.get_struct());

environment.add_transition(0, 1, Exp(0.5));
environment.add_transition(1, 0, Exp(1.0));
```

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::EnvOptions opt;                 // stage_solver defaults to "fluid"
const env::EnvAnalyzerSolution<double> res = env::solver_env(environment, opt);
```

The returned table reports performance metrics averaged over the stationary distribution of the environment. The two stages can also be analyzed independently in steady-state with `MVA`, recovering the

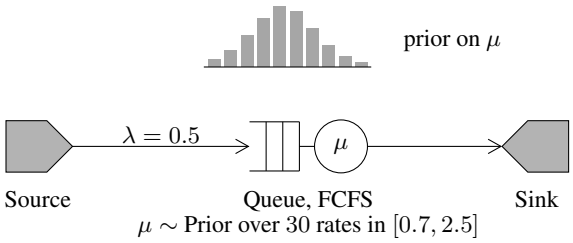

Figure 2.14: The model of Tutorial 13: an M/M/1 queue whose service rate is not known exactly but is given by a prior over 30 alternatives.

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:

```
Network model("UncertainServiceModel");
Source source(model, "Source");
Queue queue(model, "Queue", SchedStrategy::FCFS);
Sink sink(model, "Sink");
OpenClass jobClass(model, "Jobs");
source.set_arrival(jobClass, Exp(0.5));

const std::size_t numAlternatives = 30;
const double priorMean = 1.3, priorStd = 0.4;
std::vector<Distrib<double>> alternatives;
std::vector<double> priorProbs(numAlternatives);
double norm = 0.0;
for (std::size_t i = 0; i < numAlternatives; ++i) {
    const double rate = 0.7 + (2.5 - 0.7) * double(i) / double(numAlternatives - 1);
    alternatives.push_back(Exp(rate));
    const double z = (rate - priorMean) / priorStd;
    priorProbs[i] = std::exp(-0.5 * z * z);
    norm += priorProbs[i];
}
for (double& p : priorProbs) p /= norm;
queue.set_service(jobClass, prior_discrete(alternatives, priorProbs));

Routing P;
P.set(source, queue, 1.0);
P.set(queue,  sink,  1.0);
model.link(P);
```

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

```
const uq::UqStageSolver<double> stage =
    [](const qn::NetworkStruct<double>& sn) {
        mva::MvaOptions o;
        return mva::solver_mva_run_analyzer(sn, o, Matrix<double>());
    };
uq::UqOptions uopt;
```

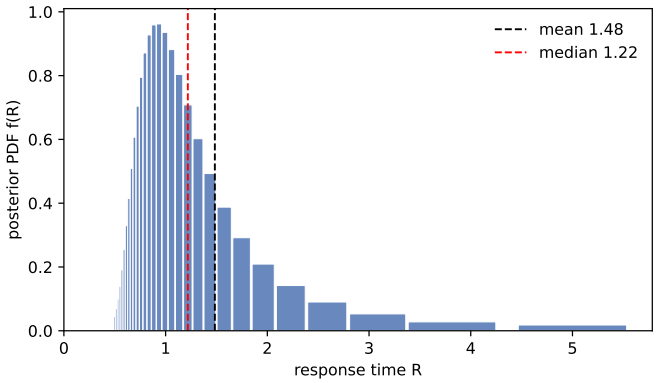

Figure 2.15: Posterior PDF of the response time R for the $M/M/1$ queue with an uncertain service rate. The distribution is right-skewed: the posterior mean (≈ 1.48) exceeds the median (≈ 1.22) because the low-service-rate alternatives contribute a long upper tail.

```
const uq::UqSolution<double> res = uq::solver_uq_run_analyzer(model, stage, uopt);
```

The solution carries the prior-weighted means and the per-alternative results the posterior distributions are formed from. The objects returned by are empirical CDFs over the requested metric. , from which one obtains expectations, modes, and quantiles in the usual way: The posterior of a metric is the alternatives' values with the posterior weights; its mean and its median are computed from that pair as above. 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: The C++ port produces the values and the weights; histogramming and plotting them is left to the caller. The resulting density (Figure 2.15) is right-skewed: slow-service alternatives in the prior contribute a 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 re-implementing 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: The cluster generators are not part of the C++ port: the three servers, their arrival split and their service rates are written out with `add_queue`, `set_service` and a routing matrix. The MVA result shows traffic split evenly across the three servers, as expected under random dispatching:

Station	JobClass	QLen	Util	RespT	ResidT	ArvR	Tput
Source	Class1	0	0	0	0	0	0.4
Server1	Class1	0.15263	0.13333	1.1448	0.38158	0.13333	0.13333
Server2	Class1	0.15263	0.13333	1.1448	0.38158	0.13333	0.13333
Server3	Class1	0.15263	0.13333	1.1448	0.38158	0.13333	0.13333

We now switch to the `Cluster` builder to introduce non-uniform multi-server queues, replacing PS

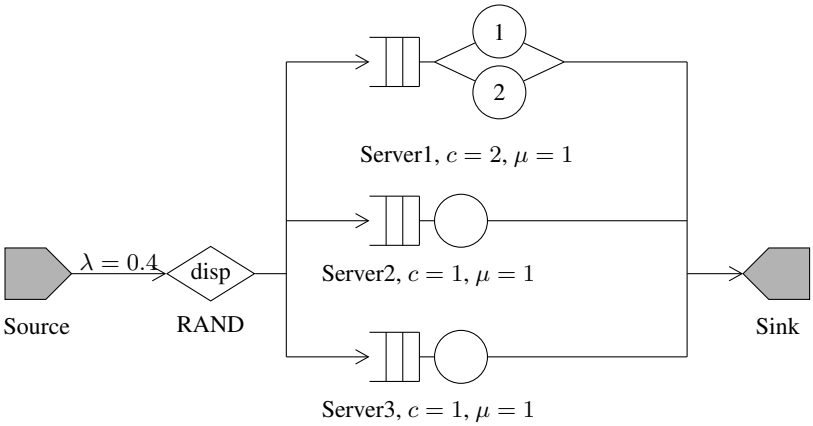

Figure 2.16: The open cluster of Tutorial 14: an arrival stream is dispatched at random over three stations, the first of which has two servers.

with FCFS and giving `Server1` two parallel cores:

```
Network model("Cluster");
Source source(model, "Source");
Sink sink(model, "Sink");
OpenClass cls(model, "Class1");
source.set_arrival(cls, Exp(0.4));
const std::vector<double> servers{2, 1, 1};   // Server1 is M/M/2
Routing P;
std::vector<std::size_t> q(servers.size());
for (std::size_t i = 0; i < servers.size(); ++i) {
    q[i] = model.add_queue("Server" + std::to_string(i + 1), SchedStrategy::FCFS);
    model.set_number_of_servers(q[i], servers[i]);
    model.set_service(q[i], cls, Exp(1.0));
    P.set(source, q[i], 1.0 / double(servers.size()));
    P.set(q[i], sink, 1.0);
}
model.link(P);
```

We then cross-check the same FCFS cluster under three simulators. `JMT` is the XML-driven Java Modelling Tools simulator; `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. All three produce statistically equivalent results on this open-class cluster and exercise the multi-server FCFS path. The C++ port's simulator is `ssa::solver_ssa`; there is no JMT or LDES bridge to cross-check against, so the comparison is between its analytical solvers.

Finally, we compare random and round-robin dispatching on a uniform PS cluster. Round-robin is non-product-form, so we drop to JMT with a small sample budget: Comparing dispatching policies is a loop over `set_routing` with `RoutingStrategy::RAND` and `RROBIN`, each followed by a solve. 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 [3]. 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*

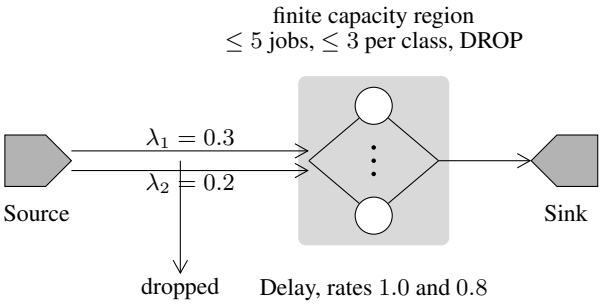

Figure 2.17: The loss network of Tutorial 15: the delay station belongs to a finite capacity region, and arrivals that would violate the global or the per-class limit are dropped.

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:

```
Network model("LossNetwork");
// Block 1: nodes
Source source(model, "Source");
Delay delay(model, "Delay");
Sink sink(model, "Sink");
// Block 2: classes
OpenClass class1(model, "Class1");
OpenClass class2(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
Routing P;
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(std::vector<std::size_t>{delay},
                 std::vector<double>{3.0, 3.0},   // per-class job caps
                 5.0,                             // region-wide cap
                 std::vector<DropStrategy>{DropStrategy::DROP,
                                           DropStrategy::DROP});
// Block 5: solution -- the Erlang fixed-point approximation is the default
nc::NcSolverOptions opt;
const mva::AvgResult<double> res = nc::solver_nc_run_analyzer(model.get_struct(), opt);
// Monte Carlo summation, which also estimates the normalizing constant
nc::NcSolverOptions mci;
mci.method = "mci";
mci.samples = 100000;
const mva::AvgResult<double> resMci = nc::solver_nc_run_analyzer(model.get_struct(), mci);
```

The region is declared in one call; there is no JMT cross-check in the C++ port, and the simulation comparison would be run with `ssa::solver_ssa`. The three solutions agree closely, confirming the product-form result against simulation:

```
AvgTable =                          % SolverNC, exact transform (default)
    Station    JobClass     QLen       Util      RespT     ResidT      ArvR       Tput
    _______    ________    _______    _______    _____     ______    _______    _______
     Delay      Class1     0.29895    0.29895        1          1     0.29895    0.29895
     Delay      Class2     0.24969    0.24969     1.25       1.25     0.19975    0.19975
AvgTableMCI =                       % SolverNC, mci (Ross-Wang)
     Delay      Class1     0.29901    0.29901        1          1     0.29901    0.29901
     Delay      Class2     0.24967    0.24967     1.25       1.25     0.19973    0.19973
lG = 0.4994                         % log-normalization constant log g(C)
AvgTableJMT =                       % SolverJMT simulation
     Delay      Class1     0.29976    0.29976    0.99627    0.99627    0.29886    0.29824
     Delay      Class2     0.25189    0.25189     1.2535     1.2535    0.20144    0.20144
```

The `SolverNC` solver recognizes the single-`Delay` DROP region as a loss network and offers three solution methods: the default `exact` transform, the `erlangfp` Erlang fixed-point approximation, 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 `SolverJMT`, whose discrete-event simulator natively supports DROP finite capacity regions; 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* [1]: 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 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 `set_service`; 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:

Queueing places are not part of the C++ port: `add_place` creates an infinite-server token container, so a place with an embedded FCFS queue cannot be expressed. Ordinary Petri-net places and transitions are supported, through `add_place` and `add_transition`. The token counts reported for the two queueing places match the exact finite-population result to within simulation error:

```
AvgTableLDES =                      % SolverLDES simulation of the QPN
    Station    JobClass     QLen      RespT     Tput
    _______    ________    ______    ______    ______
     CPU         Jobs      1.6189    1.3626    1.1881
     Think       Jobs      2.3811    2.0040    1.1881
AvgTableMVA =                       % exact SolverMVA on Delay + M/M/1
     CPU         Jobs      1.6183    1.3590    1.1908
     Think       Jobs      2.3817    2.0000    1.1908
```

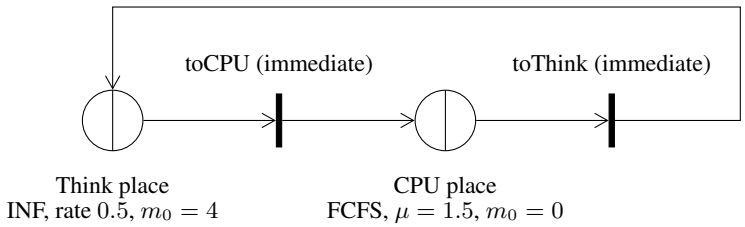

Figure 2.18: The queueing Petri net of Tutorial 16: two queueing places, drawn in the standard queueing-Petri-net shorthand as a place circle bisected by a vertical bar, exchange a single token per firing of the immediate transitions. The bar stands for the scheduling station embedded in the place, whose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

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 multi-server) 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.

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

Index

Entries are two-level. A member of an enumerated family — a solver, a node type, a scheduling or routing strategy, a distribution, a metric, a model class, an option, a solution method — is listed both under its own name and under its family, so that `FCFS` can be reached either alphabetically or through *scheduling strategies*. Page numbers point at the definition or the explanation of a term, not at every mention of it.