LINE: Queueing Analysis Algorithms

A Primer – Java Edition

Last revision: July 7, 2026

QORE Lab, Imperial College London

Contents

1 A guided tour of LINE **1**

1.1 Overview and installation . . . 1

1.2 A first model . . . 1

1.2.1 The M/M/1 queue . . . 1

1.2.2 Reading the results . . . 2

1.3 Network models . . . 2

1.3.1 Nodes, scheduling and classes . . . 2

1.3.2 Distributions . . . 2

1.3.3 A closed-network example . . . 3

1.3.4 Routing and advanced features . . . 3

1.4 Analysis and solvers . . . 3

1.4.1 Performance metrics . . . 3

1.4.2 Solvers . . . 3

1.5 Layered queueing networks . . . 4

1.6 Random environments . . . 4

1.7 Further resources . . . 4

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 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, LQNS, MAMSolver, Q-MAM, SMCSolver, and BuTools.

Add `jline.jar` (built from the `jar/` folder, or downloaded from the release page) to your classpath. All examples import from the `jline.lang` and `jline.solvers` packages. The latest releases and source are available from http://line-solver.sf.net.

LINE is released under the BSD-3 license and is maintained by the QORE lab. If you use LINE in academic work, please cite the package and the relevant method papers. LINE has been partially funded by the European Commission grants FP7-318484 (MODAClouds), H2020-644869 (DICE), H2020-825040 (RADON), and the EPSRC grant EP/M009211/1 (OptiMAM).

1.2 A first model

1.2.1 The M/M/1 queue

A LINE script has four conceptual blocks: (1) create the *nodes* (stations), (2) declare the *job classes*, (3) assign *arrival* and *service* processes, and (4) specify the *topology* (routing). The model is then handed to a *solver*. The following script builds and solves an M/M/1 queue with arrival rate $\lambda = 1$ and service rate $\mu = 2$; theory gives utilization $\rho = \lambda/\mu = 0.5$ and mean response time $1/(\mu - \lambda) = 1$s.

```
Network model = new Network("M/M/1");
// Block 1: nodes
Source source = new Source(model, "Source");
Queue queue = new Queue(model, "Queue", SchedStrategy.FCFS);
Sink sink = new Sink(model, "Sink");
// Block 2: classes
OpenClass jobclass = new OpenClass(model, "Class1");
// Block 3: arrival and service processes
source.setArrival(jobclass, new Exp(1.0)); // lambda = 1
queue.setService(jobclass, new Exp(2.0)); // mu = 2
// Block 4: topology
model.link(Network.serialRouting(source, queue, sink));
// Solve
NetworkAvgTable avgTable = new MVA(model).avgTable();
avgTable.print();
```

1.2.2 Reading the results

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

Column	Meaning
`QLen`	mean number of jobs at the station (queue length)
`Util`	utilization (busy fraction / 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

Individual rows are selected as follows:

```
avgTable.tget(queue, jobclass).print(); // by object
avgTable.tget("Queue", "Class1").print(); // by label
```

Stochastic solvers accept a `seed` for reproducibility and a `samples` budget, e.g. `new JMT(model, "seed",23000)`. Verbosity is controlled globally through `GlobalConstants`.

1.3 Network models

The `Network` object is the container for all model elements. Elements register themselves with the model passed to their constructor. The four-block workflow of Section 1.2 scales to arbitrary networks; 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: `Erlang.fitMeanAndSCV(m,scv)` matches a target mean and squared coefficient of variation.

1.3.3 A closed-network example

A machine-repairman model: three machines alternate between a working (infinite-server) delay and a two-server repair queue, solved exactly with the `CTMC` solver.

```
Network model = new Network("MRP");
Delay delay = new Delay(model, "WorkingState");
Queue queue = new Queue(model, "RepairQueue", SchedStrategy.FCFS);
queue.setNumberOfServers(2);
ClosedClass cclass = new ClosedClass(model, "Machines", 3, delay);
delay.setService(cclass, new Exp(0.5));
queue.setService(cclass, new Exp(4.0));
model.link(Network.serialRouting(delay, queue));
new CTMC(model).avgTable().print();
```

The reference station (here `delay`) passed to the closed-class constructor marks where a job is considered to have completed a cycle.

1.3.4 Routing and advanced features

Serial chains use `serialRouting` inside `link`; point-to-point links are added with `model.addLink(a,b)`, and per-node behaviour is set with `setRouting` using 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; these are covered in the full manual.

1.4 Analysis and solvers

1.4.1 Performance metrics

The primary analysis is steady-state average performance, obtained with the average-table method (all station/class metrics) or a finer-grained average method returning the metric matrices directly. System-level aggregates and, for Markovian models, the underlying state space and generator are also available (e.g. from the `CTMC` solver), together with transient averages and response-time distributions.

1.4.2 Solvers

A solver takes a model and options and returns metrics; all are invoked uniformly as *Solver(model, options)* followed by the average-table method.

Solver	Method	Best for
`CTMC`	continuous-time Markov chain	exact, small state spaces
`MVA`	mean value analysis	product-form open/closed, fast
`NC`	normalizing constant	large closed populations
`FLD`	fluid / mean-field ODE	large populations, transient
`SSA`	stochastic simulation (native)	general, any feature
`MAM`	matrix-analytic methods	MAP/PH open models
`JMT`	external simulation (JMT)	validation, general
`LN`	layered network solver	layered models (Sec. 1.5)
`LQNS`	external LQN solver	layered models
`ENV`	random-environment blending	environment models (Sec. 1.6)

As a rule of thumb, use `CTMC` for small exact models; `MVA`/`NC` for product-form networks at scale; `FLD` for very large populations or transients; `MAM` for Markovian-arrival or phase-type open models;

and SSA/JMT when a feature is not supported analytically. Running two solvers and comparing is good practice, as every solver consumes the same model object unchanged.

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). The example below is a two-tier client–server application.

```
LayeredNetwork model = new LayeredNetwork("ClientDBSystem");
Processor P1 = new Processor(model, "ClientProcessor", 1, SchedStrategy.PS);
Processor P2 = new Processor(model, "DBProcessor", 1, SchedStrategy.PS);
Task T1 = new Task(model, "ClientTask", 10, SchedStrategy.REF).on(P1);
T1.setThinkTime(Exp.fitMean(5.0));
Task T2 = new Task(model, "DBTask", Integer.MAX_VALUE, SchedStrategy.INF).on(P2);
Entry E1 = new Entry(model, "ClientEntry").on(T1);
Entry E2 = new Entry(model, "DBEntry").on(T2);
Activity A1 = new Activity(model, "ClientActivity", Exp.fitMean(1.0)).on(T1);
A1.boundTo(E1).synchCall(E2, 2.5); // 2.5 DB calls per request
Activity A2 = new Activity(model, "DBActivity", Exp.fitMean(0.8)).on(T2);
A2.boundTo(E2).repliesTo(E2);
new SolverLN(model, SolverType.MVA).avgTable().print();
```

The native LN solver decomposes the model into layers and applies a supplied per-layer solver (here MVA); the external LQNS solver may be used when available.

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 example alternates a closed model's server between a fast mode ($\mu = 4$) and a slow mode ($\mu = 1$); we assume a base closed model baseModel has already been built.

```
Environment env = new Environment("ServerModes", 2);
Network fastModel = baseModel.copy();
((Queue) fastModel.getNodeByName("Server"))
      .setService(fastModel.getClasses().get(0), new Exp(4.0));
env.addStage(0, "Fast", "operational", fastModel);
Network slowModel = baseModel.copy();
((Queue) slowModel.getNodeByName("Server"))
      .setService(slowModel.getClasses().get(0), new Exp(1.0));
env.addStage(1, "Slow", "degraded", slowModel);
env.addTransition(0, 1, new Exp(0.5));
env.addTransition(1, 0, new Exp(1.0));
new ENV(env, m -> new FLD(m)).getAvgTable().print();
```

The ENV meta-solver takes a factory that produces an inner solver for each stage and returns metrics averaged over the stationary distribution of the environment.

1.7 Further resources

The line-cli.py script solves models without writing code, reading JMT's JSIMG and LQNS's LQNX formats and printing tables, JSON, or CSV:

```
python line-cli.py solve model.jsimg --solver mva
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 MATLAB, Java, and Python manuals with their Doxygen, Javadoc, and Sphinx API references, plus the MCP getting-started guide, all linked from http://line-solver.sf.net.

For many more worked examples – additional queueing systems, layered and mixed networks, transient and distribution analysis, inference, and optimization – consult the full LINE manual for your language, which expands every topic in this primer in greater depth.