LINE: Queueing Analysis Algorithms

User manual for Kotlin

Last revision: September 22, 2025

Contents

Chapter 1

Introduction

1.1 What is LINE?

LINE is an open-source software package to analyze queueing models via analytical methods and simulation. The tool aims at simplifying the computation of performance and reliability metrics in models of systems such as software applications, business processes, or computer networks. LINE decomposes a high-level system model into one or more stochastic models, typically extended queueing networks, that are subsequently analyzed using either numerical algorithms or simulation. The stand-alone JAR version covered in this manual (https://sf.net/p/line-solver/code/ci/master/tree/jar) is based on Java/Kotlin.

A key feature of LINE is that the solver decouples the model description from the solvers used for its solution. That is, LINE implements model-to-model transformations that automatically translate the model specification into the input format (or data structure) accepted by the target solver. External solvers supported by LINE include Java Modelling Tools (JMT; http://jmt.sf.net) and LQNS (http://www.sce.carleton.ca/rads/lqns/). Native model solvers are instead based on formalisms and techniques such as:

- Continuous-time Markov chains (`CTMC`)
- Fluid ordinary differential equations (`FLUID`)
- Matrix analytic methods (`MAM`)
- Normalizing constant analysis (`NC`)
- Mean-value analysis (`MVA`)
- Stochastic Simulation Algorithms (`SSA`)

Each solver encodes a general solution paradigm and can implement both exact and approximate analysis methods. For example, the `MVA` solver implements both exact mean value analysis (MVA) and approximate

mean value analysis (AMVA). The offered methods typically differ for accuracy, computational cost, and the subset of model features they support. A special solver (AUTO) is supplied that provides an automated recommendation on which solver to use for a given model.

The above techniques can be applied to models specified in the following formats:

- LINE *modeling language*. This is a domain-specific object-oriented language designed to resemble the abstractions available in JMT's queueing network simulator (JSIM).
- *Layered queueing network models (LQNS XML format)*. LINE is able to solve a sub-class of layered queueing network models, either specified using the LINE modeling language or according to the XML metamodel of the LQNS solver.
- *JMT simulation models (JSIMg, JSIMw formats)*. LINE is able to import and solve queueing network models specified using JSIMgraph and JSIMwiz. LINE models can be exported to, and visualized with, JSIMgraph and JSIMwiz.

1.2 Obtaining the latest release

This document contains the user manual for LINE version 3.0.x, which can be obtained from:

http://line-solver.sf.net/

LINE 3.0.x has been tested using Kotlin 2.1.x with Java 8. It is also highly recommended to install Apache Maven 3.6.3 or later and IntelliJ IDEA or equivalent for development.

1.3 References

To cite the LINE solver, we recommend to reference:

- G. Casale. "Integrated Performance Evaluation of Extended Queueing Network Models with LINE", in *Proc. of WSC 2020*, ACM Press, Dec 2020.

1.4 Contact and credits

Project coordinator: Giuliano Casale, Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2AZ, London, United Kingdom. Web: http://wp.doc.ic.ac.uk/gcasale/

Please refer to the AUTHORS files in the codebase for detailed credits.

1.5 Copyright and license

1.6 Acknowledgement

LINE has been partially funded by the European Commission grants FP7-318484 (MODAClouds), H2020-644869 (DICE), H2020-825040 (RADON), and by the EPSRC grant EP/M009211/1 (OptiMAM).

Chapter 2

Getting started

2.1 Installation and support

Quick Start: This is the fastest way to get started with LINE:

1. Obtain the latest release:

   - Stable release (zip file): https://sf.net/projects/line-solver/files/latest/download

   Ensure that the files are decompressed in the installation folder.

2. Change the active directory to the `jar/` folder, then run

   ```
   mvn clean package -Pb
   ```

   This will build the JAR library (Kotlin compilation) under the `target/` folder.

2.1.1 Software requirements

Certain features of LINE depend on external tools and libraries. The recommended dependencies are:

- Kotlin 2.1.x with Java 8.
- Apache Maven 3.6.3 or later.
- IntelliJ IDEA or equivalent.

Partial Kotlin ports or interfaces to these libraries have been implemented within LINE:

- Java Modelling Tools (http://jmt.sf.net): version 1.2.4 or later. The latest version is automatically downloaded at the first call of the `JMT` solver.

- KPC-Toolbox (https://github.com/kpctoolboxteam/kpc-toolbox): version 0.3.4 or later.
- M3A (https://github.com/imperial-qore/M3A): version 1.0.0.
- BuTools (https://github.com/ghorvath78/butools): version 2.0 or later.
- Q-MAM (https://win.uantwerpen.be/~vanhoudt/tools/QBDfiles.zip).

Optional dependencies recommended to utilize all features available in LINE are as follows:

- LQNS (https://github.com/layeredqueuing/V6): version 6.2.28 or later. System paths need to be configured such that the `lqns` and `lqnsim` solvers need are available on the command line.

2.1.2 Documentation

This manual introduces the main concepts to define models in LINE and run its solvers. The document includes in particular several tables that summarize the features currently supported in the modeling language and by individual solvers. Additional resources are as follows:

- MATLAB manual: https://line-solver.sf.net/doc/LINE-matlab.pdf
- Java manual: https://line-solver.sf.net/doc/LINE-java.pdf
- Kotlin manual: https://line-solver.sf.net/doc/LINE-kotlin.pdf
- Python manual: https://line-solver.sf.net/doc/LINE-python.pdf

2.1.3 Getting help

For discussions, bug reports, new feature requests, please create a thread on the Sourceforge forums:

- General discussion: https://sf.net/p/line-solver/discussion/help/
- Bugs and issues: https://sf.net/p/line-solver/tickets/
- Feature requests: https://sf.net/p/line-solver/feature-requests/

2.2 Getting started examples

In this section, we present some examples that illustrate how to use LINE. The relevant scripts are included under the `examples/gettingstarted/` folder. The examples describe one of two main available classes of stochastic models within LINE:

- `Network` models are extended queueing networks. Typical instances are open, closed and mixed queueing networks, possibly including advanced features such as class-switching, finite capacity, priorities, non-exponential distributions, and others. Technical background on these models can be found in books such as [5, 32] or in tutorials such as [2, 31].

- `LayeredNetwork` models are layered queueing networks, i.e., models consisting of layers, each corresponding to a `Network` object, which interact through synchronous and asynchronous calls. Technical background on layered queueing networks can be found in [48].

2.2.1 Controlling verbosity

Solver verbosity may be configured at program start using, e.g.:

```
GlobalConstants.getInstance().verbose = VerboseLevel.DEBUG
```

The three available verbosity levels are:

- `VerboseLevel.SILENT`: Suppresses all solver output messages

- `VerboseLevel.STD`: Shows standard solver output messages (default)

- `VerboseLevel.DEBUG`: Shows detailed solver output including debug information

2.2.2 Model gallery

LINE includes a collection of classic, commonly occurring, queueing models under the `gallery/` folder. They include single queueing systems (e.g., $M/M/1$, $M/H_2/1$, $D/M/1$, ...), tandem queueing systems, and basic queueing networks. For example, to instantiate and estimate the mean response time for a tandem network of $M/M/1$ queues we may run

```
SolverMVA(gallery_mm1_tandem()).avgTable().print()
```

Obtaining the following printout

```
Station   JobClass  QLen       Util       RespT      ResidT     ArvR       Tput
------------------------------------------------------------------------------------
mySource  myClass   0          0          0          0          0          1.00000
Queue1    myClass   8.99916    0.90000    8.99916    8.99916    1.00000    1.00000
Queue2    myClass   8.99916    0.90000    8.99916    8.99916    1.00000    1.00000
------------------------------------------------------------------------------------
```

The examples in the gallery may also be used as templates to accelerate the definition of basic models. Example 9 shows later an example of gallery instantiation of a $M/E_2/1$ queue.

2.2.3 Example 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-serve (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.

The general structure of a LINE script consists of four blocks:

1. Definition of nodes
2. Definition of job classes and associated statistical distributions
3. Instantiation of model topology
4. Solution

For example, the following script solves the M/M/1 model

```
val model = Network("M/M/1")
// Block 1: nodes
val source = Source(model, "Source")
val queue = Queue(model, "Queue", SchedStrategy.FCFS)
val sink = Sink(model, "Sink")
// Block 2: classes
val jobclass = OpenClass(model, "Class1", 0)
source.setArrival(jobclass, Exp(1.0)) // (source,jobclass)
queue.setService(jobclass, Exp(2.0)) // (queue,jobclass)
// Block 3: topology
model.link(model.serialRouting(source, queue, sink))
// Block 4: solution
val avgTable = SolverJMT(model, "seed", 23000).avgTable()
avgTable.print()
avgTable.tget(queue, jobclass).print()
avgTable.tget("Queue", "Class1").print()
```

In the example, `source` and `sink` are arrival and departure points of jobs; `queue` is a queueing station with FCFS scheduling; `jobclass` defines an open class of jobs that arrive, get served, and leave the system; `Exp(2.0)` defines an exponential distribution with rate parameter $\lambda = 2.0$; finally, the command solves for average performance measures with JMT's 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 (`Util`); the mean response time for a

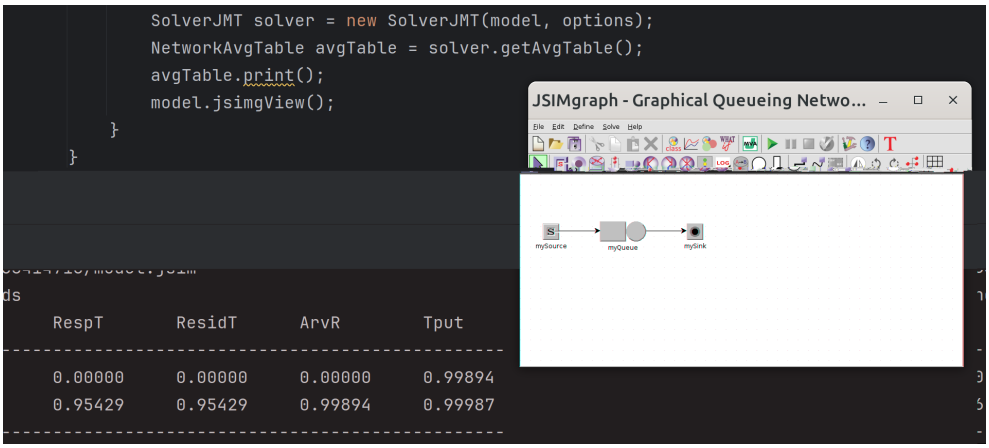

Figure 2.1: M/M/1 example in JSIMgraph

visit to the station (`RespT`); the mean residence time, i.e. the mean response time cumulatively spent at the station over all visits (`ResidT`); the mean throughput of departing jobs (`Tput`)

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Source      Class1      0           0           0           0           0           0.99894
Queue       Class1      0.95550     0.48736     0.95429     0.95429     0.99894     0.99987
-------------------------------------------------------------------------------------------
```

One can verify that this matches JMT results by first typing

```
model.jsimgView()
```

which will open the model inside JSIMgraph, as shown in Figure 2.1. From this screen, the simulation can be started using the green "play" button in the JSIMgraph toolbar. A pre-defined gallery of classic models is also available, for example

```
val model = gallery_mm1()
```

returns a M/M/1 queue with 50% utilization.

If we want to select a particular row of the `avgTable` data structure, we can use the `tget` (table get) command, for example

```
avgTable.tget("Queue", "Class1").print()
```

gives output

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Queue       Class1      0.95550     0.48736     0.95429     0.95429     0.99894     0.99987
```

```
--------------------------------------------------------------------------------
```

If we specify only `"Queue"` or `"Class1"`, `tget` will return all entries corresponding to that station or class. Moreover, the following syntax is also valid

```
avgTable.tget(queue, jobclass).print()
```

if we specify only `queue` or only `jobclass`, will return all entries corresponding to that station or class.

2.2.4 Example 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

```
val model = Network("M/G/1")
val source = Source(model, "Source")
val queue = Queue(model, "Queue", SchedStrategy.FCFS)
val 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

```
val jobclass1 = OpenClass(model, "Class1", 0)
val jobclass2 = OpenClass(model, "Class2", 0)
source.setArrival(jobclass1, Exp(0.5))
source.setArrival(jobclass2, Exp(0.5))
queue.setService(jobclass1, Erlang.fitMeanAndSCV(1.0, 1.0 / 3.0))
try {
    val fileURI = GettingStarted::class.java.getResource("/example_trace.txt")!!.toURI()
    val fileName = Paths.get(fileURI).toString()
    queue.setService(jobclass2, Replayer(fileName))
} catch (e: URISyntaxException) {
    throw RuntimeException(e)
}
```

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

```
val P = model.initRoutingMatrix()
P.set(jobclass1, jobclass1, Network.serialRouting(source, queue, sink))
P.set(jobclass2, jobclass2, Network.serialRouting(source, queue, sink))
model.link(P)
```

and solve the model with JMT

```
val avgTable: NetworkAvgTable = SolverJMT(model).avgTable()
avgTable.print()
```

which gives

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
------------------------------------------------------------------------------------------
Source      Class1      0           0           0           0           0           0.50017
Source      Class2      0           0           0           0           0           0.49114
Queue       Class1      0.86153     0.49840     1.73889     1.73889     0.50017     0.49953
Queue       Class2      0.43751     0.04918     0.85879     0.85879     0.49114     0.49064
------------------------------------------------------------------------------------------
```

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 [4] based on the trace

```
queue.setService(jobclass2, Replayer(fileName).fitAPH())
```

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

```
SolverCTMC(model,"cutoff",2,"verbose", VerboseLevel.STD).avgTable().print()
Station  JobClass  QLen        Util        RespT       ResidT      ArvR        Tput
------------------------------------------------------------------------------------
Source   Class1    0           0           0           0           0           0.44108
Source   Class2    0           0           0           0           0           0.47594
Queue    Class1    0.56714     0.44108     1.28578     1.28578     0.44108     0.44108
Queue    Class2    0.24423     0.04810     0.51316     0.51316     0.47594     0.47594
------------------------------------------------------------------------------------
```

However, we see from the comparison with JMT that the errors of `SolverCTMC` 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

```
SolverCTMC(model,"cutoff",4).avgTable().print()
```

which gives

```
-------------------------------------------------------------------------------------------
Station  JobClass  QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Source   Class1    0           0           0           0           0           0.49160
Source   Class2    0           0           0           0           0           0.49570
Queue    Class1    0.79557     0.49160     1.61832     1.61832     0.49160     0.49160
Queue    Class2    0.37532     0.05010     0.75716     0.75716     0.49570     0.49570
-------------------------------------------------------------------------------------------
```

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. `SolverMAM` uses the repetitive structure of the CTMC to exactly analyze open systems with an infinite state space, calling

```
SolverMAM(model).avgTable().print()
```

we get

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Source      Class1      0           0           0           0           0           0.50000
Source      Class2      0           0           0           0           0           0.50000
Queue       Class1      0.87649     0.50000     1.75299     1.75299     0.50000     0.50000
Queue       Class2      0.42703     0.05054     0.85406     0.85406     0.50000     0.50000
-------------------------------------------------------------------------------------------
```

2.2.5 Example 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:

```
val model = Network("MRP")
val delay = Delay(model, "WorkingState")
val queue = Queue(model, "RepairQueue", SchedStrategy.FCFS)
queue.setNumberOfServers(2)

val closedClass = ClosedClass(model, "Machines", 3, delay)
delay.setService(closedClass, Exp(0.5))
queue.setService(closedClass, Exp(4.0))
model.link(model.serialRouting(delay, queue))
val solver = SolverCTMC(model)
val ctmcAvgTable = solver.avgTable()
ctmcAvgTable.print()
```

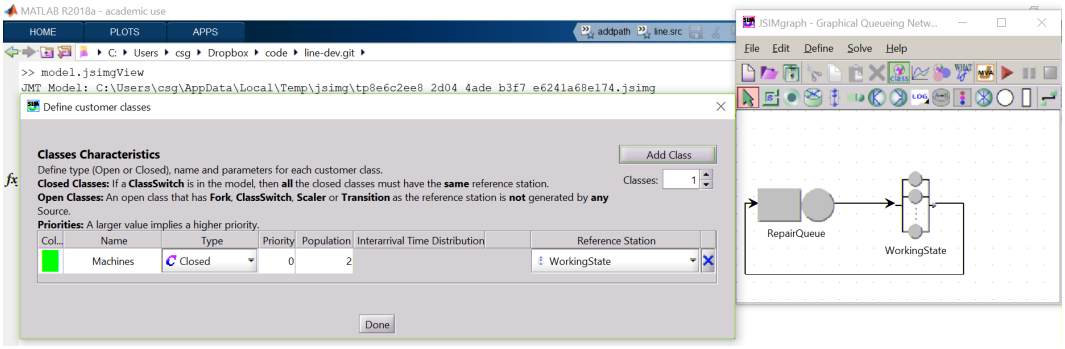

Figure 2.2: Machine interference model in JSIMgraph

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

```
Station      JobClass    QLen     Util     RespT     ResidT     ArvR         Tput
--------------------------------------------------------------------------------
WorkingState Machines    2.66484  2.66484  2.00000     2.00000     0          1.33242
RepairQueue  Machines    0.33516  0.16655  0.25154     0.25154     0          1.33242
--------------------------------------------------------------------------------
```

As before, we can inspect and analyze the model in JSIMgraph using the command

```
model.jsimgView()
```

Figure 2.2 illustrates the result, demonstrating the automated definition of the closed class.

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

```
val stateSpace = solver.stateSpace().stateSpace
stateSpace.print()
val infGen = solver.generator().infGen
infGen.print()
```

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

```
[0  1  2
 1  0  2
 2  0  1
 3  0  0]

[-8.0   8.0     0     0
  0.5  -8.5   8.0     0
    0   1.0  -5.0   4.0
    0     0   1.5  -1.5]
```

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 `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 `stateSpace` associated to the source and destination states.

On this and larger infinite generators, we may also list individual non-zero transitions as follows

```
SolverCTMC.printInfGen(infGen, stateSpace);
```

which gives

```
[ 0.0   1.0   2.0 ] -> [ 1.0   0.0   2.0 ] : 8.0
[ 1.0   0.0   2.0 ] -> [ 0.0   1.0   2.0 ] : 0.5
[ 1.0   0.0   2.0 ] -> [ 2.0   0.0   1.0 ] : 8.0
[ 2.0   0.0   1.0 ] -> [ 1.0   0.0   2.0 ] : 1.0
[ 2.0   0.0   1.0 ] -> [ 3.0   0.0   0.0 ] : 4.0
[ 3.0   0.0   0.0 ] -> [ 2.0   0.0   1.0 ] : 1.5
```

The above printout helps in matching the state transitions to their rates.

To avoid having to inspect the `stateSpace` variable to determine to which station a particular column refers to, we can alternatively use the more general invocation

```
localStateSpace.print();
```

gives

```
[0
 1
 2
 3]

[1  2
 0  2
 0  1
 0  0]
```

which automatically splits the state space into its constituent parts for each stateful node.

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

2.2.6 Example 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:

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

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

```
val jobclass = OpenClass(model, "Class1")
source.setArrival(jobclass, Exp(1.0))
queue1.setService(jobclass, Exp(2.0))
queue2.setService(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 `addLinks` function:

```
model.addLinks(arrayOf(
                arrayOf(source, lb),
                arrayOf(lb, queue1),
                arrayOf(lb, queue2),
                arrayOf(queue1, sink),
                arrayOf(queue2, sink)
))
```

At this point, all nodes are automatically configured to route jobs with equal probabilities on the outgoing links (`RoutingStrategy.RAND` policy). If we solve the model at this point, we see that the response time at the queues is around $0.66s$.

```
SolverJMT(model, "seed", 23000).avgTable().print()
```

which gives

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Source      Class1      0           0           0           0           0           1.01349
Queue1      Class1      0.31612     0.24682     0.65411     0.65411     0.50144     0.50100
Queue2      Class1      0.33403     0.25076     0.68406     0.34203     0.50446     0.50413
-------------------------------------------------------------------------------------------
```

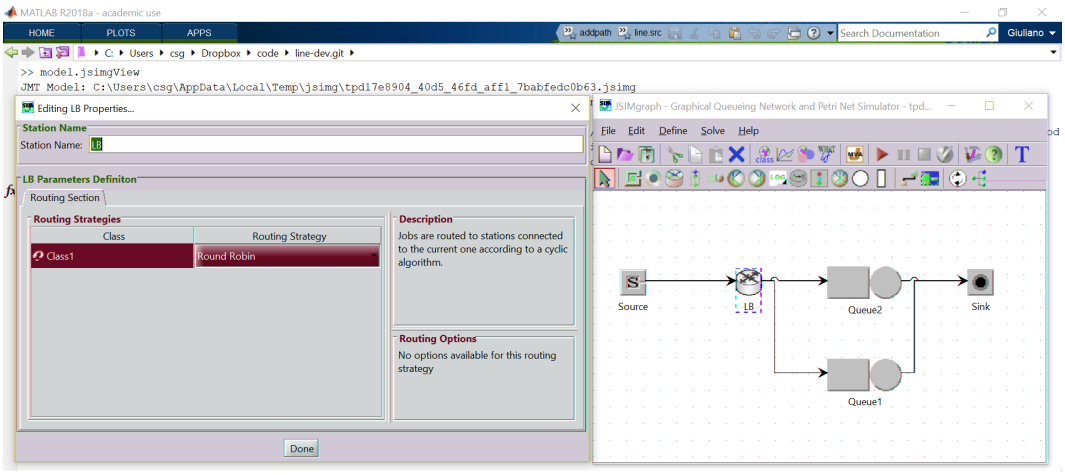

Figure 2.3: Load-balancing model

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.reset()
lb.setRouting(jobclass, RoutingStrategy.RROBIN)
```

A representation of the model at this point is shown in Figure 2.3.

Lastly, we run again JMT and find that round-robin produces a visible decrease in response times, which are now around $0.56s$.

```
SolverJMT(model, "seed", 23000).avgTable().print()
```

which gives

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
------------------------------------------------------------------------------------------
Source      Class1      0           0           0           0           0           1.00887
Queue1      Class1      0.30429     0.26118     0.58482     0.58482     0.50290     0.50526
Queue2      Class1      0.29282     0.24397     0.57293     0.28647     0.50496     0.50526
------------------------------------------------------------------------------------------
```

2.2.7 Example 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

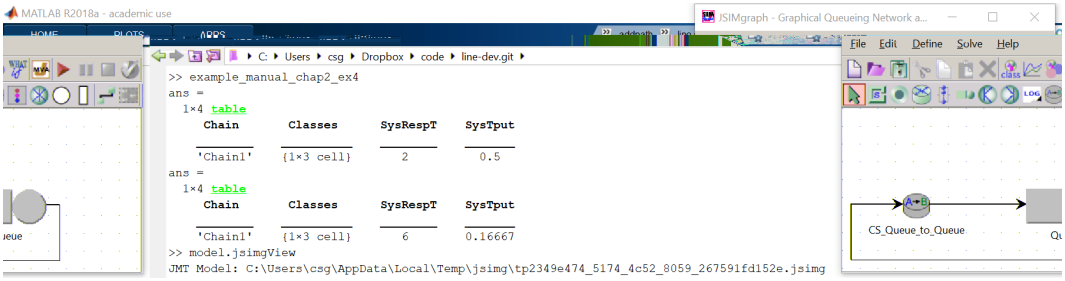

Figure 2.4: Re-entrant lines as an example of class-switching

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 by 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.

```
val model = Network("RL")
val queue = Queue(model, "Queue", SchedStrategy.FCFS)

val jobClass1 = ClosedClass(model, "Class1", 1, queue)
val jobClass2 = ClosedClass(model, "Class2", 0, queue)
val jobClass3 = ClosedClass(model, "Class3", 0, queue)

queue.setService(jobClass1, Erlang.fitMeanAndOrder(1, 2))
queue.setService(jobClass2, Erlang.fitMeanAndOrder(2, 2))
queue.setService(jobClass3, Erlang.fitMeanAndOrder(3, 2))

val P = model.initRoutingMatrix()
P.set(jobClass1, jobClass2, queue, queue, 1.0)
P.set(jobClass2, jobClass3, queue, queue, 1.0)
P.set(jobClass3, jobClass1, queue, queue, 1.0)
model.link(P)
```

The corresponding JMT model is shown in Figure 2.4, where it can be seen that the class-switching rule is automatically enforced by introduction of a `ClassSwitch` node in the network.
We can now simulate the performance indexes for the different classes, for example using LINE's normalizing constant solver (`SolverNC`)

```
SolverNC(model).avgTable().print()
```

gives

```
Station     JobClass    QLen        Util        RespT       ResidT      ArvR        Tput
--------------------------------------------------------------------------------------
```

```
Queue       Class1      0.16667     0.16667     1.00000     0.33333     0.16667     0.16667
Queue       Class2      0.33333     0.33333     2.00000     0.66667     0.16667     0.16667
Queue       Class3      0.50000     0.50000     3.00000     1.00000     0.16667     0.16667
--------------------------------------------------------------------------------------------
```

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

```
SolverNC(model).avgSysTable().print()
```

which gives

```
Chain           JobClasses              SysRespT             SysTput
------------------------------------------------------------------------------
Chain0          (Class1 Class2 Class3)  2.0000000000000000   0.5000000000000000
------------------------------------------------------------------------------
```

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

We see that the throughput of the chain is 0.5, which means that LINE is counting every departure from the queue in any class as a completion for the whole chain. This is incorrect for our model since we want to count completions only when jobs depart in `Class3`. To require this behavior, we can tell to the solver that passages for classes 1 and 2 through the reference station should not be counted as completions

```
jobClass1.setCompletes(false)
jobClass2.setCompletes(false)
```

This modification then gives the correct chain throughput, matching the one of `Class3` alone

```
SolverNC(model).avgSysTable().print()
```

```
Chain           JobClasses              SysRespT             SysTput
------------------------------------------------------------------------------
Chain1          (Class1 Class2 Class3)  6.0000000000000010   0.1666666666666667
------------------------------------------------------------------------------
```

2.2.8 Example 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 hit 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:

```
val model = Network("QNC")
// Block 1: nodes
val clientDelay = Delay(model, "Client")
val cacheNode = Cache(model, "Cache", 1000, 50, ReplacementStrategy.LRU)
val cacheDelay = Delay(model, "CacheDelay")
```

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

```
val clientClass = ClosedClass(model, "ClientClass", 1, clientDelay, 0)
val hitClass = ClosedClass(model, "HitClass", 0, clientDelay, 0)
val missClass = ClosedClass(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.setService(clientClass, Immediate())
cacheDelay.setService(hitClass, Exp.fitMean(0.2))
cacheDelay.setService(missClass, Exp.fitMean(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

```
cacheNode.setRead(clientClass, Zipf(1.4, 1000))
```

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

```
cacheNode.setHitClass(clientClass, hitClass)
cacheNode.setMissClass(clientClass, missClass)
```

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

```
val P = model.initRoutingMatrix()
P.set(clientClass, clientClass, clientDelay, cacheNode, 1.0)
```

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

```
P.set(hitClass, hitClass, cacheNode, cacheDelay, 1.0)
P.set(missClass, missClass, cacheNode, cacheDelay, 1.0)
```

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

```
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:

```
val ssaAvgTable = SolverSSA(model, "samples", 20000, "seed", 1, "verbose", true).avgTable()
```

The above script produces the following result

```
SSA samples:   20000
SolverSSA analysis (method: default, lang: java) completed.
Station     JobClass     QLen        Util        RespT       ResidT      ArvR        Tput
-------------------------------------------------------------------------------------------
Client      ClientClass  0           0           1.0e-08     0           0           2.91621
CacheDelay  HitClass     0.47617     0.47617     0.20000     0.16524     0           2.38083
CacheDelay  MissClass    0.52383     0.52383     1.00000     0.17380     0           0.52383
-------------------------------------------------------------------------------------------
```

The departing flows from the `cacheDelay` 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`

```
AvgTable ssaAvgTablePara = new SolverSSA(model)
    .options()
    .method("parallel")
    .samples(20000)
    .seed(1)
    .build()
    .avgTable();
```

This gives us a rather similar result, when run on a dual-core machine

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

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.

2.2.9 Example 7: 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 queueing station.

```
val model = Network("Model")

val node = arrayOfNulls<Node>(2)
node[0] = Delay(model, "Delay")
node[1] = 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.

```
val jobclass = arrayOfNulls<JobClass>(1)
jobclass[0] = ClosedClass(model, "Class1", 5, node[0] as Station, 0)

(node[0] as Delay).setService(jobclass[0]!!, Exp(1.0))
(node[1] as Queue).setService(jobclass[0]!!, Exp(0.5))

model.link(Network.serialRouting(node[0], node[1]))
```

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 `getCdfRespT` method

```
val RDfluid = SolverFluid(model).cdfRespT()
val RDsim = SolverJMT(model, "seed", 23000, "samples", 10000).cdfRespT()
```

The returned data structures, `RDfluid` and `RDsim`, are Matrix arrays where element (i,r) describes the response times at station i for class r. . 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.

which produces the graph shown in Figure **??**.

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

```
// Calculate percentiles from CDF data
Matrix cdf = RDsim[1][0]; // Station 2, Class 1
double prc95 = 0, prc99 = 0;
for (int i = 0; i < cdf.getNumRows(); i++) {
    if (cdf.get(i, 0) < 0.95) {
        prc95 = cdf.get(i, 1);
    }
    if (cdf.get(i, 0) < 0.99) {
        prc99 = cdf.get(i, 1);
    }
}
System.out.println("95th percentile: " + prc95);
```

```
System.out.println("99th percentile: " + prc99);
```

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.

2.2.10 Example 8: Optimizing a performance metric

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

In this example, we use the `jcobyla` optimizer, which can be imported by including the following dependency in `pom.xml`:

```
<dependency>
  <groupId>de.xypron.jcobyla</groupId>
  <artifactId>jcobyla</artifactId>
  <version>1.3</version>
</dependency>
```

We will now define a queueing network model, called `routingModel`, which accepts a parameter describing a routing probability

```
Calcfc objFun = new Calcfc() {
    @Override
    public double compute(int n, int m, double[] x, double[] con) {
        return routingModel.apply(x[0]) ;
    }
};
```

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

```
val model = Network("LoadBalCQN")
// Block 1: nodes
val delay = Delay(model, "Think")
val queue1 = Queue(model, "Queue1", SchedStrategy.PS)
val queue2 = Queue(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
val cclass = ClosedClass(model, "Job1", 16, delay)
delay.setService(cclass, Exp(1))
queue1.setService(cclass, Exp(0.75))
queue2.setService(cclass, Exp(0.50))
```

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

```
val P = model.initRoutingMatrix()
P.set(cclass, queue1, delay, 1.0)
P.set(cclass, queue2, delay, 1.0)
model.link(P)
```

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:

```
val routingModel = { p: Double ->
    // Block 4: solution
    P.set(cclass, delay, queue1, p)
    P.set(cclass, delay, queue2, 1 - p)
    model.reset()
    model.link(P)
    val solver = SolverMVA(model, "exact")
    solver.avgSysRespT()[0] as Double
}
```

Lastly, we optimize the function we defined

```
val p = doubleArrayOf(0.5)
Cobyla.findMinimum(routingModel, 1, 0, p, 0.5, 1.0e-8, 0, 10000)

println("Optimal p: ${p[0]}")
```

We are now ready to run the example. The execution returns the optimal value 0.6104880200336327.

2.2.11 Example 9: 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.

```
val model = gallery_merl1()
```

We now extract 50,000 samples from simulation based on the underpinning continuous-time Markov chain

```
val solver = SolverCTMC(model, "cutoff", 150, "seed", 23000)
val sa = solver.sampleSysAggr(100000)
```

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.

```
// Display structure of sampled aggregated data
println("Sample aggregate data structure:")
println("State data points: ${sa.getNumRows()}")
```

```
println("Event data available: ${sa.getNumCols() > 1}")
```

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`).

```
// Access event data
Event event1 = sa.getEvent(1);
System.out.println("Event 1 - Node: " + event1.getNode() + ", Event: " + ...
event1.getEventType());
Event event2 = sa.getEvent(2);
System.out.println("Event 2 - Node: " + event2.getNode() + ", Event: " + ...
event2.getEventType());
```

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

```
// Filter events for departures from queue node
int queueIndex = model.getNodeIndex(queue);
boolean[] filtEvent = sa.filterEvents(queueIndex, EventType.DEP);
```

Followed by a calculation of the time series of inter-departure times

```
// Calculate inter-departure times
double[] departureTimes = sa.getFilteredEventTimes(filtEvent);
double[] interDepTimes = new double[departureTimes.length - 1];
for (int i = 1; i < departureTimes.length; i++) {
    interDepTimes[i-1] = departureTimes[i] - departureTimes[i-1];
}
```

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

```
// Calculate squared coefficient of variation
double mean = Arrays.stream(interDepTimes).average().orElse(0.0);
double variance = Arrays.stream(interDepTimes).map(x -> Math.pow(x - mean, ...
2)).average().orElse(0.0);
double SCVdEst = variance / (mean * mean);
System.out.println("SCV estimate: " + SCVdEst);
```

which evaluates to 0.8750. Using Marshall's exact formula for the $GI/G/1$ queue [34], we get a theoretical value of 0.8750.

2.2.12 Example 10: 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:

```
val model = LayeredNetwork("ClientDBSystem")
```

Next, we define the processors and tasks:

```
// Create processors
val P1 = Processor(model, "ClientProcessor", 1, SchedStrategy.PS)
val P2 = Processor(model, "DatabaseProcessor", 1, SchedStrategy.PS)

// Create tasks
val T1 = Task(model, "ClientTask", 10, SchedStrategy.REF).on(P1)
T1.setThinkTime(Exp.fitMean(5.0)) // 5-second think time
val T2 = Task(model, "DatabaseTask", Int.MAX_VALUE, SchedStrategy.INF).on(P2)
```

Now we define the entries that represent service interfaces:

```
val E1 = Entry(model, "ClientEntry").on(T1)
val E2 = Entry(model, "DatabaseEntry").on(T2)
```

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

```
// Client activity: processes request and calls database
val A1 = Activity(model, "ClientActivity", Exp.fitMean(1.0)).on(T1)
A1.bound_to(E1).synch_call(E2, 2.5) // 2.5 database calls on average

// Database activity: processes database request
val A2 = Activity(model, "DatabaseActivity", Exp.fitMean(0.8)).on(T2)
A2.bound_to(E2).replies_to(E2)
```

Now we solve the layered network using the LN solver with MVA applied to each layer: The output shows the performance metrics for each node in the layered network:

```
val solver = SolverLN(model, SolverType.MVA)
val avgTable = solver.avgTable()
avgTable.print()
```

The output shows the performance metrics for each node in the layered network:

```
Node        NodeType    QLen        Util        RespT       ResidT      ArvR        Tput
P1          Processor   NaN         0.99595     NaN         NaN         NaN         NaN
P2          Processor   NaN         0.04146     NaN         NaN         NaN         NaN
T1          Task        23.46825    0.66407     NaN         1.09193     NaN         13.28141
T2          Task        8.67822     0.33188     NaN         0.55610     NaN         13.27534
T3          Task        0.12049     0.04146     NaN         0.02000     NaN         6.21891
E1          Entry       23.46825    NaN         1.76700     NaN         NaN         13.28141
E2          Entry       8.67822     NaN         0.65371     NaN         NaN         13.27534
E3          Entry       0.12049     NaN         0.01938     NaN         NaN         6.21891
AS1         Activity    23.43886    0.66407     1.76479     1.09193     NaN         13.28141
AS2         Activity    8.66725     0.33188     0.65288     0.55610     NaN         13.27534
AS3         Activity    0.12438     0.04146     0.02000     0.02000     NaN         6.21891
```

Chapter 3

Network models

Throughout this chapter, we discuss the specification of `Network` models, which are extended queueing networks. LINE currently supports open, closed, and mixed networks with non-exponential service and arrivals, and state-dependent routing. All solvers support the computation of basic performance metrics, while some more advanced features are available only in specific solvers. Each `Network` model requires in input a description of the nodes, the network topology, and the characteristics of the jobs that circulate within the network. In output, LINE returns performance and reliability metrics.

The default metrics supported by all solvers are as follows:

- Mean queue-length (`QLen`). This is the mean number of jobs residing at a node when this is observed at a random instant of time.

- Mean utilization (`Util`). For nodes that serve jobs, this is the mean fraction of time the node is busy processing jobs. In both single-server and multi-server nodes, this is a number normalized between 0 and 1, corresponding to 0% and 100%. In infinite-server nodes, the utilization is set by convention equal to the mean queue-length, therefore taking the interpretation of the mean number of jobs in execution at the station.

- Mean response time (`RespT`). This is the mean time a job spends traversing a node within a network. If the node is visited multiple times, the response time is the time spent for a single visit to the node.

- Mean residence time (`ResidT`). This is the total time a job accumulates, on average, to traverse a node within a network. If the node is visited multiple times, the residence time is the time accumulated overall visits to the node prior to returning to the reference station or arriving to a sink.

- Mean throughput (`Tput`). This is the mean departure rate of jobs completed at a resource per time unit. Typically, this matches the mean arrival rate, unless the node switches the class of the jobs in which case the arrival rate of a class may not match its departure rate.

Table 3.1: Nodes available in `Network` models.

Node	Description
`Cache`	A node to switch job classes based on hits/misses in its cache
`ClassSwitch`	A node to switch job classes based on a static probability matrix
`Delay`	A station where jobs spend time without queueing
`Fork`	A node that forks jobs into tasks
`Join`	A node that joins sibling tasks into the original job
`Logger`	A node that logs passage of jobs
`Queue`	A node where jobs queue and receive service
`Router`	A node that routes jobs to other nodes
`Sink`	Exit point for jobs in open classes
`Source`	Entry point for jobs in open classes

The above metrics refer to the performance characteristics of individual nodes. Response times and throughputs can also be system-wide, meaning that they can describe end-to-end performance during the visit to the network. In this case, these metrics are called *system* metrics.

3.1 Network object definition

3.1.1 Creating a network and its nodes

A queueing network can be described in LINE using the `Network` class constructor with a unique string identifying the model name:

```
val model: Network = Network("myModel")
```

The returned object of the `Network` class offers functions to instantiate and manage resource *nodes* (stations, delays, caches, ...) visited by jobs of several types (*classes*).

A node is a resource in the network that can be visited by a job. A node must have a unique name and can either be *stateful* or *stateless*, the latter meaning that the node does not require state variables to determine its state or actions. If jobs visiting a stateful node can be required to spend time in it, the node is also said to be a *station*. A list of nodes available in `Network` models is given in Table 3.1.

We now provide more details on each of the nodes available in `Network` models.

Queue node. A `Queue` specifies a queueing station from its name and scheduling strategy, e.g.

```
val queue: Queue = Queue(model, "Queue1", SchedStrategy.FCFS)
```

specifies a first-come first-serve queue. It is alternatively possible to instantiate a queue using the `QueueingStation` constructor, which is merely an alias for `Queue`.

Queueing stations have by default a single server. The `setNumberOfServers` method can be used to instantiate multi-server stations.

Valid scheduling strategies are specified within the `SchedStrategy` static class and include:

- First-come first-serve (`SchedStrategy.FCFS`)
- Infinite-server (`SchedStrategy.INF`)
- Processor-sharing (`SchedStrategy.PS`)
- Service in random order (`SchedStrategy.SIRO`)
- Discriminatory processor-sharing (`SchedStrategy.DPS`)
- Generalized processor-sharing (`SchedStrategy.GPS`)
- Shortest expected processing time (`SchedStrategy.SEPT`)
- Shortest job first (`SchedStrategy.SJF`)
- Head-of-line priority (non-preemptive) (`SchedStrategy.HOL`)
- Polling (`SchedStrategy.POLLING`)

If a strategy requires class weights, these can be specified directly as an argument to the `setService` function or using the `setStrategyParam` function, see later the description of DPS scheduling for an example.

Delay node. Delay stations, also called infinite server stations, may be instantiated either as objects of `Queue` class, with the `SchedStrategy.INF` scheduling strategy, or using the following specialized constructor

```
val delay: Delay = Delay(model, "ThinkTime")
```

As for queues, for readability it is possible to instantiate delay nodes using the `DelayStation` class which is an alias for the `Delay` class.

Source and Sink nodes. As seen in the M/M/1 getting started example, these nodes are mandatory elements for the specification of open classes. Their constructor only requires a specification of the unique name associated to the nodes:

```
val source: Source = Source(model, "Source")
val sink: Sink = Sink(model, "Sink")
```

Fork and Join nodes. The fork and join nodes are currently available only for the `JMT` solver. The `Fork` splits an incoming job into a set of sibling tasks, sending out one task for each outgoing link. These tasks inherit the class of the original job and are served as normal jobs until they are reassembled at a `Join` station.

Their specification of Fork and Join nodes only requires the name of the node

```
val fork: Fork = Fork(model, "Fork")
val join: Join = Join(model, "Join", fork)
```

The number of tasks sent by a `Fork` on each output link can be set using the `setTasksPerLink` method of the `fork` object. To enable effective analytical approximations, presently LINE requires that every join node is bound to a specific fork node, although specific solvers will ignore this information (e.g., JMT).

Also note that the routing probabilities out of the `Fork` node need to be set to 1.0 towards every other node connected to the `Fork`. For example, a `Fork` sending jobs in class 1 to nodes A, B and C, cannot send jobs in class 2 only to A and B: it must send them to all three connected nodes A, B and C. A new fork node visited only by class-2 jobs needs to be created in order to send that class of jobs only to A and B.

After splitting a job into tasks, LINE takes the convention that visit counts refer to the average number of passages at the target resources for the original job, scaled by the number of tasks. For example, if a job is split into two tasks at a fork node, each visiting respectively nodes A and B, the average visit count at A and B will be 0.5.

ClassSwitch node. This is a stateless node to change the class of a transiting job based on a static probabilistic policy. For example, it is possible to specify that all jobs belonging to class 1 should become of class 2 with probability 1.0, or that a transiting job of class 2 should become of class 1 with probability 0.3. This example is instantiated as follows

```
val cs: ClassSwitch = ClassSwitch(model,"ClassSwitchPoint", Matrix("[0.0, 1.0; 0.3, 0.7]"))
```

Note that the argument of the `Matrix` constructor is a string specifying a matrix in MATLAB format, i,e., where the semi-colon separates rows. Alternatively, it is also possible to use Python format, e.g.,

```
val cs: ClassSwitch = ClassSwitch(model,"ClassSwitchPoint", Matrix("[[0.0, 1.0], [0.3, ...
0.7]]"))
```

Cache node. This is a stateful node to store one or more items in a cache of finite size, for which it is possible to specify a replacement policy. The *cache* constructor requires the total cache capacity and the number of items that can be referenced by the jobs in transit, e.g.,

```
val cacheNode: Cache = Cache(model, "Cache1", nitems, capacity, ReplacementStrategy.LRU)
```

If the capacity is a scalar integer (e.g., `15`), then it represents the total number of items that can be cached and the value cannot be greater than the number of items. Conversely, if it is a vector of integers (e.g., Matrix("[10,5]")) then the node is a list-based cache, where the vector entries specify the capacity of each list. We point to [25] for more details on list-based caches and their replacement policies.

Available replacement policies are specified within the `ReplacementStrategy` static class and include:

- First-in first-out (`ReplacementStrategy.FIFO`)
- Random replacement (`ReplacementStrategy.RR`)
- Least-recently used (`ReplacementStrategy.LRU`)
- Strict first-in first-out (`ReplacementStrategy.SFIFO`)

Upon cache hit or cache miss, a job in transit is switched to a user-specified class. More details are given later in Section 3.1.5.

Router node. This node is able to route jobs according to a specified `RoutingStrategy`, which can either be probabilistic or not (e.g., round-robin). Upon entering a `Router`, a job neither waits nor receives service; it is instead directly forwarded to the next node according to the specified routing strategy. A `Router` can be instantiated as follows:

```
val router: Router = Router(model, "RouterNode")
```

An example of use of this node is given in Section 2.2.6. Routing strategies need to be specified for each class using the `setRouting` method and valid choices are as follows

- Random routing (`RoutingStrategy.RAND`)
- Round robin (`RoutingStrategy.RROBIN`)
- Probabilistic routing (`RoutingStrategy.PROB`)
- Join-the-shortest-queue (`RoutingStrategy.JSQ`)

For example, assume that `oclass` is a class of jobs. In order to route jobs in this class with equal probabilities to every outgoing link we set

```
router.setRouting(oclass, RoutingStrategy.RAND)
```

It should be noted that `setRouting` is also available for all other nodes such as queueing stations, delays, etc. Therefore, the added value of the `Router` node is the ability to represent certain system elements that centralize the routing logic, such as load balancers.

Logger node. A logger node is a node that closely resembles the logger node available in the JSIMgraph simulator within JMT. At present, models that include this element can only be solved using the `JMT` solver.

A `Logger` node records information about passing jobs in a `csv` file, such as the timestamp of passage and general information about the jobs. The node can be instantiated as follows

```
val logger = Logger(model, "LoggerNode", "logfile.csv")
```

The following methods can be used to specify the information that needs to be stored in the `csv` file

- `setStartTime`: record a timestamp for the wallclock time when the simulation started.
- `setJobID`: record a unique id for the passing job.
- `setJobClass`: record the class of the passing job.
- `setTimestamp`: record a timestamp for the simulated time when the job passed in the logger.
- `setTimeSameClass`: record the time elapsed since the last passage of a job of the same class.
- `setTimeAnyClass`: record the time elapsed since the last passage of a job of any class.

Each method can be called either with a single `true` or `false` argument, to enable or disable the recording of the corresponding information, e.g.

```
logger.setJobClass(true)
```

The routing behavior of jobs can be set up as explained for regular nodes such as queues or delay stations.

3.1.2 Advanced node parameters

Scheduling parameters

Upon setting service distributions at a station, one may also specify scheduling parameters such as weights as additional arguments to the `setService` function. For example, if the node implements discriminatory processor sharing (`SchedStrategy.DPS`), the command

```
queue.setService(class2, Cox2.fitMeanAndSCV(0.2,10.0), 5.0)
```

assigns a weight 5.0 to jobs in class 2. The default weight of a class is 1.0.

Finite buffers

The functions `setCapacity` and `setChainCapacity` of the `Station` class are used to place constraints on the number of jobs, total or for each chain, that can reside within a station. Note that LINE does not allow one to specify buffer constraints at the level of individual classes unless chains contain a single class, in which case `setChainCapacity` is sufficient for the purpose.

For example,

```
cqn_threeclass_hyperl()
delay.setChainCapacity(Matrix("[1,1]"))
model.refreshCapacity()
```

creates an example model with two chains and three classes (specified in `cqn_threeclass_hyperl.m`) and requires the second station to accept a maximum of one job in each chain. Note that if we were to ask for a higher capacity, such as , which exceeds the total job population in chain 2, LINE would have automatically reduced the value 7 to the chain 2 job population (2). This automatic correction ensures that functions that analyze the state space of the model do not generate unreachable states.

The `refreshCapacity` function updates the buffer parameterizations, performing appropriate sanity checks. Since `cqn_threeclass_hyperl` has already invoked a solver prior to our changes, the requested modifications are materially applied by LINE to the network only after calling an appropriate `refreshStruct` function, see the sensitivity analysis section. If the buffer capacity changes were made before the first solver invocation on the model, then there would not be the need for a `refreshCapacity` call, since the internal representation of the `Network` object used by the solvers is still to be created.

Cyclic polling

In the polling scheduling policy, the server cyclically visits the input buffer of each job class, processing jobs from that queue for a finite amount of time before switching to the next buffer. This scheduling policy further requires to specify the polling type at the station, chosen among:

- Exhaustive (`PollingType.EXHAUSTIVE`), where the server moves to the next input buffer only after finishing all jobs in the current buffer, including new arrivals during the service period;
- Gated (`PollingType.GATED`), where the number of served jobs for a buffer equals the jobs therein at the instant where the server switched to processing that buffer. Thus, newly arrived jobs during the service period will be processed at the next period.
- K-Limited (`PollingType.KLIMITED`), where the number of jobs processed is a constant K specified by the user. If the queue empties before k jobs have been served, the server will move on and attend the next buffer.

To specify a polling type at a queue, we may write for instance

```
queue.setPollingType(PollingType.EXHAUSTIVE)
// or for K-Limited with K=2
queue.setPollingType(PollingType.KLIMITED, 2)
```

3.1.3 Job classes

Jobs travel within the network placing service demands at the stations. The demand placed by a job at a station depends on the class of the job. Jobs in *open classes* arrive from the external world and, upon

completing the visit, leave the network. Jobs in *closed classes* start within the network and are forbidden to ever leave it, perpetually cycling among the nodes.

Open classes

The constructor for an open class only requires the class name and the creation of special nodes called `Source` and `Sink`

```
val source: Source = Source(model, "Source")
val sink: Sink = Sink(model, "Sink")
```

Sources are special stations holding an infinite pool of jobs and representing the external world. Sinks are nodes that route a departing job back into this infinite pool, i.e., into the source. Note that a network can include at most a single `Source` and a single `Sink`.

Once source and sink are instantiated in the model, it is possible to instantiate open classes using

```
val class1: OpenClass = OpenClass(model, "Class1")
```

LINE does not require explicitly associating source and sink with the open classes in their constructors, as this is done automatically. However, the LINE language requires explicitly creating these nodes since the routing topology needs to indicate the arrival and departure points of jobs in open classes. However, if the network does not include open classes, the user will not need to instantiate a `Source` and a `Sink`.

Closed classes

To create a closed class, we need instead to indicate the number of jobs that start in that class (e.g., 5 jobs) and the *reference station* for that class (e.g., `queue`), i.e.:

```
val class2: ClosedClass = ClosedClass(model, "Class2", 5, queue)
```

The reference station indicates a point in the network used to calculate certain performance indexes, called *system performance indexes*. The end-to-end response time for a job in an open class to traverse the system is an example of a system performance index (system response time). The reference station of an open class is always automatically set by LINE to be the . Conversely, the reference station needs to be indicated explicitly in the constructor for closed classes since the point at which a class job completes execution depends on the semantics of the model.

LINE also supports a special class of jobs, called *self-looping jobs*, which perpetually loop at the reference station, remaining in their class. The following example shows the syntax to specify a self-looping job, which is identical to closed classes but there is no need later to specify routing information.

```
val model: Network = Network("model")
// Block 1: nodes
val delay: Delay = Delay(model, "Delay")
val queue: Queue = Queue(model, "Queue1", SchedStrategy.FCFS)
// Block 2: classes
```

```
val cclass: ClosedClass = ClosedClass(model, "Class1", 10, delay, 0)
val slclass: SelfLoopingClass = SelfLoopingClass(model, "SLC", 1, queue, 0)
delay.setService(cclass, Exp(1.0))
queue.setService(cclass, Exp(1.5))
queue.setService(slclass, Exp(1.5))
// Block 3: topology
val P: Matrix = model.initRoutingMatrix()
P.set(cclass, Matrix("[0.7,0.3;1.0,0]"))
model.link(P)
```

Note that any routing information specified for the self-looping class will be ignored.

Mixed models

LINE also accepts models where a user has instantiated both open and closed classes. The only requirement is that, if two classes communicate by means of a class-switching mechanism, then the two classes must either be all closed or all open. In other words, classes in the same chain must either be both closed or open. Furthermore, for all closed classes in the same chain, it is required for the reference station to be the same.

Class priorities

If a class has a priority, with 0 representing the highest priority, this can be specified as an additional argument to both `OpenClass` and `ClosedClass`, e.g.,

```
val class2: ClosedClass = ClosedClass(model, "Class2", 5, queue, 0)
```

In `Network` models, priorities are intended as hard priorities and the only supported priority scheduling strategy (`SchedStrategy.HOL`) is non-preemptive. Weight-based policies such as DPS and GPS may be used, as an alternative, to prevent starvation of jobs in low-priority classes.

Class switching

In LINE, jobs can switch their class while they travel between nodes (including self-loops on the same node). For example, this feature can be used to model queueing properties such as re-entrant lines in which a job visiting a station a second time may require a different average service demand than at its first visit.

A chain defines the set of reachable classes for a job that starts in the given class r and over time changes class. Since class switching in LINE does not allow a closed class to become open, and vice-versa, chains can themselves be classified into *open chains* and *closed chains*, depending on the classes that compose them.

Jobs in open classes can only switch to another open class. Similarly, jobs in closed classes can only switch to a closed class. Thus, class switching from open to closed classes (or vice-versa) is forbidden. More details about class-switching are given in Section 3.1.5.

Reference station

Before we have shown that the specification of classes requires choosing a reference station. In LINE, reference stations are properties of chains, thus if two closed classes belong to the same chain they must have the same reference station. This avoids ambiguities in the definition of the completion point for jobs within a chain.

For example, the system throughput for a chain is defined as the sum of the arrival rates at the reference station for all classes in that chain. That is, the solver counts a return to the reference station as a completion of the visit to the system. In the case of open chains, the reference station is always the `Source` and the system throughput corresponds to the rate at which jobs arrive at the sink `Sink`, which may be seen as the arrival rate seen by the infinite pool of jobs in the external world. If there is no class switching, each chain contains a single class, thus per-chain and per-class performance indexes will be identical.

Reference class

Occasionally, it is possible to encounter situations where a job needs to change class while remaining inside the same station. In this case, LINE modifies the network automatically to introduce a class-switching node for the job to route out of the station and immediately return to it in the new class.

One complication of the approach is that, by departing the node and returning to it, the job visits the station one additional time, affecting the visit count to the station and therefore performance metrics such as the residence time. To cope with this issue, LINE offers a method for the class objects, called `setReferenceClass`, that allows users to specify whether the visit of that class to the reference station should be considered upon computing the residence times across the network for the chain to which the class belongs. By default, all classes traversing the reference station are used in the visit count calculation.

3.1.4 Routing strategies

Probabilistic routing

Jobs travel between nodes according to the network topology and a routing strategy. Typically a queueing network will use a probabilistic routing strategy (`RoutingStrategy.PROB`), which requires specifying routing probabilities among the nodes. The simplest way to specify a large routing topology is to define the routing probability matrix for each class, followed by a call to the `link` function. This function will automatically add certain nodes to the network to ensure the correct class switching for jobs moving between stations (`ClassSwitch` elements).

In the running case, we may instantiate a routing topology as follows:

```
val P: Matrix = model.initRoutingMatrix()
P.set(class1,source,queue,1.0)
P.set(class1,queue,queue,0.3); // self-loop with probability 0.3
P.set(class1,queue,delay,0.7); // transition with probability 0.7
P.set(class1,delay,sink,1.0)
P.set(class2,delay,queue,1.0); // note: closed class jobs start at delay
```

```
P.set(class2,queue,delay,1.0)
model.link(P)
```

When used as arguments to a cell array or matrix, class, and node objects will be replaced by a corresponding numerical index. Normally, the indexing of classes and nodes matches the order in which they are instantiated in the model and one can therefore specify the routing matrices using this property. In this case, we would have

```
P = model.initRoutingMatrix()
P.set(class1, Matrix("[0,1,0,0; 0,.3,.7,0; 0,0,0,1; 0,0,0,0]"))
P.set(class2, Matrix("[0,0,0,0; 0,0,1,0; 0,1,0,0; 0,0,0,0]"))
model.link(P)
```

Where needed, the functions return the numerical index associated with a node name, for example . Class and node names in a network *must be unique*. The list of names already assigned to nodes in the network can be obtained with the functions of the `Network` class.

It is also important to note that the routing matrix in the last example is specified between *nodes*, instead of between just stations or stateful nodes, which means that for example elements such as the need to be explicitly considered in the routing matrix. The only exception is that `ClassSwitch` elements do not need to be explicitly instantiated in the routing matrix, provided that one uses the `link` function to instantiate the topology. Note that the routing matrix assigned to a model can be printed on the screen in a human-readable format using the `printRoutingMatrix` function, e.g.,

```
model.printRoutingMatrix()
```

prints

```
Delay [Class1] => Queue1 [Class1] : Pr=1.0
Delay [Class2] => Queue1 [Class2] : Pr=0.001
Queue1 [Class1] => Queue1 [Class1] : Pr=0.3
Queue1 [Class1] => Source [Class1] : Pr=0.7
Queue1 [Class2] => Source [Class2] : Pr=1.0
Source [Class1] => Sink [Class1] : Pr=1.0
Source [Class2] => Queue1 [Class2] : Pr=1.0
Sink [Class2] => Source [Class2] : Pr=1.0
```

Other routing strategies

The above routing specification style is only for models with probabilistic routing strategies between every pair of nodes. A different style should be used for scheduling policies that do not require to explicit routing probabilities, as in the case of state-dependent routing. Currently supported strategies include:

- Round robin (`RoutingStrategy.RROBIN`). This is a deterministic strategy that sends jobs to outgoing links in a cyclic order.

- Random routing (`RoutingStrategy.RAND`). This is equivalent to a standard probabilistic strategy that for each class assigns identical values to the routing probabilities of all outgoing links. When a target is invalid its probability is kept to zero, e.g., random routing will not send a job in a closed class to a sink.

- Join-the-Shortest-Queue (`RoutingStrategy.JSQ`). This is a non-probabilistic strategy that sends jobs to the destination with the smallest total number of jobs in it (either queueing or receiving service). If multiple stations have the same total number of jobs, then the destination is chosen at random with equal probability.

For the above policies, the function `addLink` should be first used to specify pairs of connected nodes

```
model.addLink(queue, queue); //self-loop
model.addLink(queue, delay)
```

Then an appropriate routing strategy should be selected at every node, e.g.,

```
queue.setRouting(class1,RoutingStrategy.RROBIN)
```

assigns round robin among all outgoing links from the `queue` node.

A model could also include both classes with probabilistic routing strategies and classes that use round-robin or other non-probabilistic strategies. To instantiate routing probabilities in such situations one should then use, e.g.,

```
queue.setRouting(class1,RoutingStrategy.PROB)
queue.setProbRouting(class1, queue, 0.7)
queue.setProbRouting(class1, delay, 0.3)
```

where `setProbRouting` assigns the routing probabilities to the two links.

Routing probabilities for Source and Sink nodes

In the presence of open classes, and in mixed models with both open and closed classes, one needs only to specify the routing probabilities *out* of the source. The probabilities out of the sink can all be set to zero for all classes and destinations (including self-loops). The solver will take care of adjusting these inputs to create a valid routing table.

Simplified definition of tandem and cyclic topologies

Tandem networks are open queueing networks with a serial topology. LINE provides functions that ease the definition of tandem networks of stations with exponential service times. For example, the getting started Example 1 on the M/M/1 queue illustrates a simplified way to specify a serial routing topology, i.e.,

```
model.link(Network.serialRouting(source,queue,sink))
```

In a similar fashion, we can also rapidly instantiate a tandem network consisting of stations with PS and INF scheduling as follows

```
Matrix lambda = new Matrix("[10,20]"); // lambda(r) - arrival rate of class r
Matrix D = new Matrix("[11,12; 21,22]"); // D(i,r) - class-r demand at station i (PS)
Matrix Z = new Matrix("[91,92;93,94]"); // Z(i,r)  - class-r demand at station i (INF)
Network modelPsInf = Network.tandemPsInf(lambda,D,Z);
```

The above snippet instantiates an open network with two queueing stations (PS), two delay stations (INF), and exponential distributions with the given inter-arrival rates and mean service times. The `Network.tandemPs`, `Network.tandemFcfs`, and `Network.tandemFcfsInf` functions provide static constructors for networks with other combinations of scheduling policies, namely only PS, only FCFS, or FCFS and INF.

A tandem network with closed classes is instead called a cyclic network. Similar to tandem networks, LINE offers a set of static constructors:

- `Network.cyclicPs`: cyclic network of PS queues
- `Network.cyclicPsInf`: cyclic network of PS queues and delay stations
- `Network.cyclicFcfs`: cyclic network of FCFS queues
- `Network.cyclicFcfsInf`: cyclic network of FCFS queues and delay stations

These functions only require replacing the arrival rate vector `A` by a vector `N` specifying the job populations for each of the closed classes, e.g.,

```
// Create cyclic PS network
Matrix N = new Matrix("[2,1]"); // job populations
Matrix D = new Matrix("[[11,12],[21,22]]"); // service demands
Network modelPsInf = Network.cyclicPs(N, D);
```

3.1.5 Class switching

Depending on the specified probabilities, a job will be able to switch its class only among a subset of the available classes. Each subset is called a *chain*. Chains are computed in LINE as the weakly connected components of the routing probability matrix of the network when this is seen as an undirected graph. The function `model.getChains()` produces the list of chains for the model, inclusive of a list of their composing classes.

The definition of class switching in a model is integrated in the specification of the routing between stations as described in the next subsection.

Probabilistic class switching

In models with class switching and probabilistic routing at all nodes, a routing matrix is required for each possible pair of source and target classes. For instance, suppose that in the previous example the job in the

closed class `class2` switches into a new closed class (`class3`) while visiting the `queue` node. We can specify this routing strategy as follows:

```
val class3: ClosedClass = ClosedClass(model, "Class3", 0, queue, 0)
val P: Matrix = model.initRoutingMatrix()
P.set(class1,source,queue,1.0)
P.set(class1,queue,queue,0.3); // self-loop with probability 0.3
P.set(class1,queue,delay,0.7); // transition with probability 0.7
P.set(class1,delay,sink,1.0)
P.set(class2,class3,delay,queue,1.0); // note: closed class jobs start at delay
P.set(class3,class2,queue,delay,1.0)
model.link(P)
```

Importantly, LINE assumes that a job switches class an instant *after* leaving a station, thus the performance metrics of a class at the node refer to the class that jobs had upon arrival to that node.

Class switching with non-probabilistic routing strategies

In the presence of non-probabilistic routing strategies, such as round-robin or join-the-shortest-queue, one may need to manually specify the details of the class switching mechanism. This can be done through addition to the network topology of `ClassSwitch` nodes.

The constructor of the `ClassSwitch` node requires a probability matrix C such that the lement in row r and column s is the probability that a job of class r arriving into the node switches to class s during the visit. For example, in a 2-class model, the following node will switch all visiting jobs into class 2

```
// Block 1: nodes
// ...
val csnode: ClassSwitch = ClassSwitch(model, "ClassSwitch 1")
// Block 2: classes
val jobclass1: OpenClass = OpenClass(model, "Class1", 0)
val jobclass2: OpenClass = OpenClass(model, "Class2", 0)
// ...
// Block 3: topology
val C: ClassSwitchMatrix = csnode.initClassSwitchMatrix()
C.set(jobclass1, jobclass2, 1.0)
C.set(jobclass2, jobclass2, 1.0)
csnode.setClassSwitchingMatrix(C)
```

Note that for a network with M stations, up to M^2 `ClassSwitch` elements may be required to implement class-switching across all possible links, including self-loops.

Cache-based class-switching

An advanced feature of LINE available for example within the `Cache` node, is that the class-switching decision can dynamically depend on the state of the node (e.g., cache hit/cache miss). However, in order to statically determine chains, LINE requires that every class-switching node declares the pair of classes that can potentially communicate with each other via a switch. This is called the *class-switching mask* and it is automatically computed. The boolean matrix returned by the `model.getClassSwitchingMask` function

provides this mask, which has an entry in row r and column s set to true only if jobs in class r can switch into class s at some node in the network.

Upon cache hit or cache miss, a job in transit is switched to a user-specified class, as specified by the `setHitClass` and `setMissClass`, so that it can be routed to a different destination based on whether it found the item in the cache or not. The `setRead` function allows the user to specify a discrete distribution (e.g., `Zipf`, `DiscreteSampler`) for the frequency at which an item is requested. For example,

```
val refModel: Zipf = Zipf(0.5,nitems)
cacheNode.setRead(initClass, refModel)
cacheNode.setHitClass(initClass, hitClass)
cacheNode.setMissClass(initClass, missClass)
```

Here `initClass`, `hitClass`, and `missClass` can be either open or closed instantiated as usual with the `OpenClass` or `ClosedClass` constructors.

3.1.6 Service and inter-arrival time processes

A number of statistical distributions are available to specify job service times at the stations and inter-arrival times from the station. The class `Markovian` offers distributions that are analytically tractable, which are defined using absorbing Markov chains consisting of one or more states (*phases*) and called phase-type distributions. They include as special cases the following distributions:

- Exponential distribution: `Exp`(λ), where λ is the rate of the exponential
- n-phase Erlang distribution: `Erlang`(α, n), where α is the rate of each of the n exponential phases
- 2-phase hyper-exponential distribution: `HyperExp`$(p, \lambda_1, \lambda_2)$, that returns an exponential with rate λ_1 with probability p, and an exponential with rate λ_2 otherwise.
- n-phase hyper-exponential distribution: `HyperExp`(p, λ), that builds a n-phase hyper-exponential from a rate vector $\lambda = [\lambda_1, \ldots, \lambda_n]$ and phase selection probabilities $p = [p_1, \ldots, p_n]$.
- 2-phase Coxian distribution: `Coxian`(μ_1, μ_2, ϕ_1), which assigns phases μ_1 and μ_2 to the two rates, and completion probability from phase 1 equal to ϕ_1 (the probability from phase 2 is $\phi_2 = 1.0$).
- n-phase Coxian distribution: `Coxian`(μ, ϕ), which builds an arbitrary Coxian distribution from a vector $\mu = [\mu_1, \ldots, \mu_n]$ of n rates and a completion probability vector $\phi = [\phi_1, \ldots, \phi_n]$ with $\phi_n = 1.0$.
- n-phase acyclic phase-type distribution: `APH`(α, T), which defines an acyclic phase-type distribution with initial probability vector $\alpha = [\alpha_1, \ldots, \alpha_n]$ and transient generator T.

For example, given mean $\mu = 0.2$ and squared coefficient of variation SCV=10, where SCV=variance/μ^2, we can assign to a node a 2-phase Coxian service time distribution with these moments as

```
queue.setService(class2, Cox2.fitMeanAndSCV(0.2,10.0))
```

where `Cox2` is a static class to fit 2-phase `Coxian` distributions. Inter-arrival time distributions can be instantiated in a similar way, using `setArrival` instead of `setService` on the `Source` node. For example, if the `Source` is node 3 we may assign the inter-arrival times of class 2 to be exponential with mean 0.1 as follows

```
source.setArrival(class2, Exp.fitMean(0.1))
```

Is it also possible to plot the structure of a phase-type distribution using `Markovian.plot` static method.

Non-Markovian distributions are also available, but typically they can restrict the available solvers to the `JMT` simulator. They include the following distributions:

- Deterministic distribution: `Det`(μ) assigns probability 1.0 to the value μ.
- Uniform distribution: `Uniform`(a, b) assigns uniform probability $1/(b-a)$ to the interval $[a, b]$.
- Gamma distribution: `Gamma`(α, k) assigns a gamma density with shape α and scale k.
- Pareto distribution: `Pareto`(α, k) assigns a Pareto density with shape α and scale k.

Lastly, we discuss two special distributions. The `Disabled` distribution can be used to explicitly forbid a class to receive service at a station. This may be useful to declare in models with sparse routing matrices to debug the model specification. Performance metrics for disabled classes will be set to `Double.NaN`.

Conversely, the `Immediate` class can be used to specify instantaneous service (zero service time). Normally, solvers will replace zero service times with small positive values ($\varepsilon =$`GlobalConstants.FineTol`).

Fitting a distribution

The `fitMeanAndSCV` function is available for all distributions that inherit from the `Markovian` class. This function provides exact or approximate matching of the first two moments, depending on the theoretical constraints imposed by the distribution. For example, an Erlang distribution with SCV=0.75 does not exist, because in a n-phase Erlang it must be SCV=$1/n$. In a case like this, `Erlang.fitMeanAndSCV(1,0.75)` will return the closest approximation, e.g., a 2-phase Erlang (SCV=0.5) with unit mean. The Erlang distribution also offers a function `fitMeanAndOrder`(μ, n), which instantiates a n-phase Erlang with given mean μ.

In distributions that are uniquely determined by more than two moments, `fitMeanAndSCV` chooses a particular assignment of the residual degrees of freedom other than mean and SCV. For example, `HyperExp` depends on three parameters, therefore it is insufficient to specify mean and SCV to identify the distribution. Thus, `HyperExp.fitMeanAndSCV` automatically chooses to return a probability of selecting phase 1 equal to 0.99. Compared to other choices, this particular assignment corresponds to a higher probability mass in the tail of the distribution. `HyperExp.fitMeanAndSCVBalanced` instead assigns p in a two-phase hyper-exponential distribution so that $p/\mu_1 = (1-p)/\mu_2$.

Inspecting and sampling a distribution

To verify that the fitted distribution has the expected mean and SCV it is possible to use the `getMean` and `getSCV` functions, e.g.,

```
val dist: Exp = Exp(1)
System.out.println(dist.getMean())
System.out.println(dist.getSCV())
```

prints

```
1.0
1.0
```

Moreover, the `sample` function can be used to generate values from the obtained distribution, e.g. we can generate 3 samples as

```
System.out.println(Arrays.toString(dist.sample(3)));
```

The `evalCDF` and `evalCDFInterval` functions return the cumulative distribution function at the specified point or within a range, e.g.,

```
System.out.println(dist.evalCDFInterval(2, 5));
System.out.println(dist.evalCDF(5) - dist.evalCDF(2));
```

For more advanced uses, the distributions of the `Markovian` class also offer the possibility to obtain the standard (D_0, D_1) representation used in the theory of Markovian arrival processes by means of the `getRepresentation` function [5].

Load-dependent service

A queueing station i is called *load-dependent* whenever its service rate is a function of the number n_i of resident jobs at the station, summed across the ones in service and the ones in the waiting buffer. For example, a multi-server station with c identical servers, each with processing rate μ, may be shown to behave similarly to a single-server load-dependent station where the service rate is $\mu(n_i) = \mu\alpha(n_i) = \mu \min(n_i, c)$.

LINE presently supports *limited load-dependence* [11], meaning that it is possible to specify the form of the load-dependent service up to a finite range of n_i. As such, the support is currently limited to closed models, which are guaranteed to have a finite population at all times.

To specify a load-dependence service for a queueing station over the range $n_i \in [1, N]$ it is sufficient to call the `setLoadDependence` method, passing a vector of size N in its input with the scaling factor values for each n_i. For example, to instantiate a c-server node we write

```
val LD: Matrix = Matrix(1, N)
for (int i=0; i<N; i++) {
    LD.set(i, Math.min(i + 1, c))
}
queue.setLoadDependence(LD)
```

where the i-th element of the vector argument of `setLoadDependence` is the scaling factor $\alpha(n_i)$. It is assumed by default that $\alpha(0) = 1$.

Class-dependent service

A generalization of load-dependent service is *class-dependent* service, where the service rate is now a function of the vector $n_i = [n_{i,1}, \ldots, n_{i,R}]$, where $n_{i,r}$ is the current number of class-r jobs at station i.

LINE supports class-dependence in the MVA solver, provided that this is specified as a function handle. The solver implicitly assumes that the function is smooth and defined also for fractional values of $n_{i,r}$. For example, in a two-class model we may write

```
// Class-dependent service using lambda expression
queue.setClassDependence(ni -> Math.min(ni[1], c));
```

applies a multiserver-type only to class-2 jobs, but not to the others.

Switchover times

In multiclass models, queueing stations alternate over time the processing of jobs of different classes. In some real-world situations, overheads may arise when a server needs to be reconfigured to process a different class of service. Such overheads are referred to as *switchover times*.

LINE supports switchover times in queueing stations. For example, to configure the overhead to begin processing *jobclass2* jobs after *jobclass1* we may write

```
queue.setSwitchover(jobclass1, jobclass2, Exp(1))
```

A special case arises when the station uses (deterministic) polling scheduling. In this situation, the switchover time specifies the time for moving from the input buffer of class i to the input buffer of class $i + 1$ mod R, where R is the total number of input classes to the queue. Using this fact, we need only to specify the switchover time after each class, e.g., the switchover time to begin processing *jobclass2* jobs after *jobclass1* is set as

```
queue.setSwitchover(jobclass1, Exp(1))
```

Temporal dependent processes

It is sometimes useful to specify the statistical properties of a *time series* of service or inter-arrival times, as in the case of systems with short- and long-range dependent workloads. When the model is stochastic, we refer to these as situations where one specifies a *process*, as opposed to only specifying the *distribution* of the service or inter-arrival times. In LINE processes include the 2-state Markov-modulated Poisson process (`MMPP2`) and empirical traces read from files (`Replayer`).

For the latter, LINE assumes that empirical traces are supplied as text files (ASCII), formatted as a column of numbers. Once specified, the `Replayer` object can be used as any other distribution. This means

that it is possible to run a simulation of the model with the specified trace. However, analytical solvers will require tractable distributions from the `Markovian` class.

3.2 Internals

In this section, we discuss the internal representation of the `Network` objects used within the LINE solvers. By applying changes directly to this internal representation it is possible to considerably speed up the sequential evaluation of models.

3.2.1 Representation of the model structure

For efficiency reasons, once a user requests to solve a `Network`, LINE calls internally generate a static representation of the network structure using the `refreshStruct` function. This function returns a representation object that is then passed on to the chosen solver to parameterize the analysis.

The representation used within LINE is the `NetworkStruct` class, which describes an extended multi-class queueing network with class-switching and acyclic phase-type (APH) service times. APH generalizes known distributions such as Coxian, Erlang, Hyper-Exponential, and Exponential. The representation can be obtained as follows

```
val sn: NetworkStruct = model.getStruct()
```

The class provides idiomatic Kotlin access to the internal representation with improved type safety and nullable annotations. Table 3.2 lists the main properties available in the Kotlin API.

Table 3.2: `NetworkStruct` static properties (JAR version)

Field	Type	Description
`nstations`	`int`	Number of stations in the network
`nstateful`	`int`	Number of stateful nodes in the network
`nnodes`	`int`	Number of nodes in the network
`nclasses`	`int`	Number of classes in the network
`nclosedjobs`	`int`	Total number of jobs in closed classes
`nchains`	`int`	Number of chains in the network
`nodenames`	`List⟨String⟩`	Name of node i (accessed as `nodenames.get(i)`)
`classnames`	`List⟨String⟩`	Name of class r (accessed as `classnames.get(r)`)
`nodetype`	`List⟨NodeType⟩`	Type of node i (e.g., `NodeType.Queue`)
`njobs`	`Matrix`	Number of jobs in class r (infinity for open classes)
`nservers`	`Matrix`	Number of servers at station i
`classprio`	`Matrix`	Priority of class r (0 = highest priority)
`connmatrix`	`Matrix`	true if node i can route jobs to node j
`isstation`	`Matrix`	true if node i is a station
`isstateful`	`Matrix`	true if node i is a stateful node
`isstatedep`	`Matrix`	true if node i has state-dependent section ($s = 0$: input, $s = 1$: service, $s = 2$: routing)

Continued on next page

Table 3.2 – *Continued from previous page*

Field	Type	Description
sched	Map⟨Station, SchedStrategy⟩	Scheduling strategy at station i (e.g., SchedStrategy.PS)
schedparam	Matrix	Parameter for class r scheduling strategy at station i
mu	Map⟨Station, Map⟨JobClass, Matrix⟩⟩	Service or arrival rate in phase k for class r at station i, with mu = NaN if Disabled and mu = 10^7 if Immediate
phi	Map⟨Station, Map⟨JobClass, Matrix⟩⟩	Completion probability in phase k for class r at station i
pie	Map⟨Station, Map⟨JobClass, Matrix⟩⟩	Entry probability in phase k for class r at station i
proc	Map⟨Station, Map⟨JobClass, MatrixCell⟩⟩	Matrix representation of the class r service process at station i. For Markovian distributions, this follows the standard (D_0, D_1) representation of Markovian Arrival Processes"
proctype	Map⟨Station, Map⟨JobClass, ProcessType⟩⟩	Service or arrival process type at station i for class r (e.g., ProcessType.HYPEREXP)
phases	Matrix	Number of phases for service process of class r at station i
refstat	Matrix	Index of reference station for class r
refclass	Matrix	Index of reference class for chain c
nodeparam	Map⟨Node, NodeParam⟩	Parameters for local variable instantiation at stateful node i
nodeToStation	Matrix	Map from node index to station index (-1 if not a station)
nodeToStateful	Matrix	Map from node index to stateful index (-1 if not stateful)
stationToNode	Matrix	Map from station index to corresponding node index
stationToStateful	Matrix	Map from station index to stateful index (-1 if not stateful)
statefulToNode	Matrix	Map from stateful index to corresponding node index
statefulToStation	Matrix	Map from stateful index to station index (-1 if not a station)
droprule	Map⟨Station, Map⟨JobClass, DropStrategy⟩⟩	Drop rule for class r at station i (e.g., DropStrategy.WaitingQueue)
routing	Map⟨Node, Map⟨JobClass, RoutingStrategy⟩⟩	Routing strategy for class r upon departing node i (e.g., RoutingStrategy.JSQ)
rtorig	Map⟨JobClass, Map⟨JobClass, Matrix⟩⟩	Probability matrix specified by the user at model definition time for class switch from class r to s
lst	Map⟨Station, Map⟨JobClass, SerializableFunction⟩⟩	Laplace-Stieltjes transform of the service or arrival distribution for class r at station i
state	Map⟨StatefulNode, Matrix⟩	Current state of stateful node i. May be initially empty and updated by solver during execution
stateprior	Map⟨StatefulNode, Matrix⟩	Prior probability for states of node i
space	Map⟨StatefulNode, Matrix⟩	The state space for stateful node i. May be initially empty and updated by solver during execution
sync	Map⟨Integer, Sync⟩	Data structure specifying a synchronization among nodes
gsync	Map⟨Integer, GlobalSync⟩	Data structure specifying a global synchronization among nodes
cdscaling	Map⟨Station, SerializableFunction⟩	Class-dependent scaling when station i contains n_{ir} jobs in class r
phasessz	Matrix	Number of state vector elements used to describe phase
phaseshift	Matrix	Position shift to read phase element in state

Continued on next page

Table 3.2 – *Continued from previous page*

Field	Type	Description
nvars	Matrix	Number of local state variables for stateful node i. Position r describes state variables for class-r service; position $r = nclasses + s$ for class-s routing; positions after $2 \times nclasses$ are used for class-independent variables
isslc	Matrix	true if class r is a self-looping class
fj	Matrix	true if forked tasks from fork node f join at node j
lldscaling	Matrix	Load-dependent scaling when station i contains n_i jobs, including the ones in service
varsparam	Matrix	Variable parameters for stateful nodes
rtfun	SerializableFunction⟨Pair⟨Map⟨Node, Matrix⟩, Map⟨Node, Matrix⟩⟩, Matrix⟩	State-dependent routing table function given initial and final state Maps. Returns routing probabilities as in rt

The Kotlin API provides the same computed properties as the Java version with idiomatic Kotlin data types and improved type safety.

Table 3.3: NetworkStruct computed properties (JAR version)

Field	Type	Description
chains	Matrix	true if class r is in chain c, or false otherwise
classcap	Matrix	Maximum buffer capacity available to class r at station i
cap	Matrix	Total capacity at station i
rates	Matrix	Service rate of class r at station i (or arrival rate if i is a)
scv	Matrix	Squared coefficient of variation of class r service times at station i (or inter-arrival times if station i is a)
visits	Map⟨Integer, Matrix⟩	Number of visits that a job in chain c pays to stateful node i in class r (accessed as visits.get(c))
nodevisits	Map⟨Integer, Matrix⟩	Number of visits that a job in chain c pays to node i in class r (accessed as nodevisits.get(c))
rt	Matrix	Probability of routing from stateful node i to j, switching class from r to s where $idx_{ir} = (i-1) \times nclasses + r$
rtnodes	Matrix	Same as rt, but i and j are nodes, not necessarily stateful ones
inchain	Map⟨Integer, Matrix⟩	Indexes of classes in chain c (accessed as inchain.get(c))
sync	Map⟨Integer, Sync⟩	Data structure specifying a synchronization s among nodes
space	Map⟨StatefulNode, Matrix⟩	The t-th state in the state space (or a portion thereof). This field may be initially empty and updated by the solver during execution
state	Map⟨StatefulNode, Matrix⟩	Current state of stateful node i. This field may be initially empty and updated by the solver during execution
csmask	Matrix	true if class r can switch into class s at some node

For advanced nodes, such as Cache and Transition, additional parameters are specified under the nodeparam cell array for the corresponding node. Tables 3.4 and 3.5 illustrate the Kotlin implementation of these parameters.

Table 3.4: TransitionNodeParam fields (Kotlin version)

Field	Type	Description
enabling	List⟨Matrix⟩	Enabling condition matrices for modes at transition node

Continued on next page

Table 3.4 – *Continued from previous page*

Field	Type	Description
inhibiting	List⟨Matrix⟩	Inhibiting condition matrices for modes at transition node
modenames	List⟨String⟩	Names of modes at transition node
nmodes	int	Number of modes for transition node
nmodeservers	Matrix	Number of servers for each mode
timing	List⟨TimingStrategy⟩	Firing timing strategy for each mode (e.g., TimingStrategy.IMMEDIATE)
firingprocid	Map⟨Mode, ProcessType⟩	Firing process type for each mode (e.g., ProcessType.HYPEREXP)
firingproc	Map⟨Mode, MatrixCell⟩	Matrix representation of firing process for each mode
firingpie	Map⟨Mode, Matrix⟩	Entry probabilities for firing process phases
firingphases	Matrix	Number of phases for firing process of each mode
firing	List⟨Matrix⟩	Firing output matrices for each mode
firingprio	Matrix	Firing priority for each mode
fireweight	Matrix	Firing weight for each mode

Table 3.5: CacheNodeParam fields (Kotlin version)

Field	Type	Description
accost	Array⟨Array⟨Matrix⟩⟩	Access cost matrices for moving items between lists
hitclass	Matrix	Class switching specification for cache hits
itemcap	Matrix	Item capacity for each list
missclass	Matrix	Class switching specification for cache misses
nitems	Int	Number of items in the cache
pread	Map⟨Int, List⟨Double⟩⟩	Probability distributions for reading items by class
replacestrat	ReplacementStrategy	Replacement policy (e.g., ReplacementStrategy.LRU)
actualhitprob	Matrix	Actual hit probabilities (computed)
actualmissprob	Matrix	Actual miss probabilities (computed)

As shown in the tables, internally to LINE there is an explicit differentiation between properties of nodes, stations, and stateful nodes. This distinction has impact in particular over routing and class-switching mechanisms, and also allows solvers to better differentiate between different kinds of nodes.

In some cases, one may want to access some properties of nodes that are contained in NetworkStruct fields that are however referenced by station or stateful node index. To help this and similar situations, the NetworkStruct class also provides static methods to quickly convert the indexing of nodes, stations, and stateful nodes, which is used in referencing its data structures:

- nodeToStateful
- nodeToStation
- stationToNode
- stationToStateful
- statefulToNode

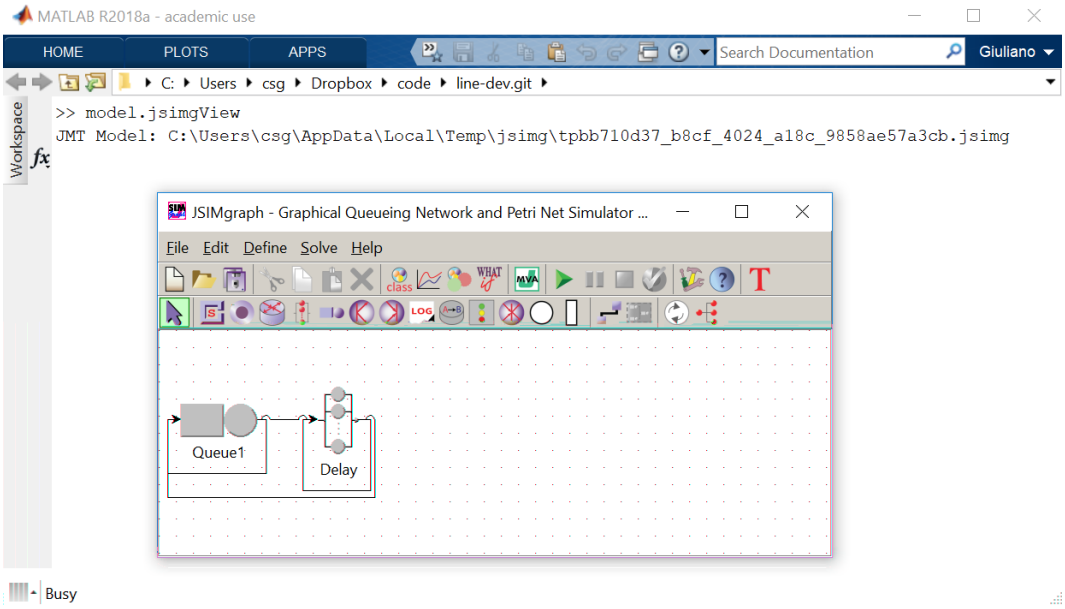

Figure 3.1: `jsimgView` function

As an example, we can determine the portion of the `nodevisits` field that refers to stateful nodes in chain $c = 1$ as follows

```
int c = 0; // class index (0-based)
Matrix V = Matrix.zeros(sn.nstateful, 1);
NetworkStruct sn = model.getStruct(); // NetworkStruct object
for (int ind = 0; ind < sn.nnodes; ind++) {
    if (sn.isstateful[ind]) {
        int isf = sn.nodeToStateful[ind];
        V.set(isf, 0, sn.nodevisits[c].get(ind, 1));
    }
}
```

3.3 Debugging and visualization

JSIMgraph is the graphical simulation environment of the JMT suite. LINE can export models to this environment for visualization purposes using the command

```
model.jsimgView()
```

An example is shown in Figure 3.1. Using a related function, `jsimwView`, it is also possible to export the model to the JSIMwiz environment, which offers a wizard-based interface.

Another way to debug a LINE model is to transform it into a graph object, i.e., Another way to debug a LINE model is to transform it into a graph object, i.e.,

```
// Display graph edges
Graph G = model.getGraph();
System.out.println("Nodes: " + G.getNodes());
System.out.println("Edges: " + G.getEdges());
```

3.4 Model import and export

LINE offers a number of scripts to import external models into `Network` object instances that can be analyzed through its solvers. The available scripts are as follows:

- `JMT2LINE` imports a JMT simulation model (`.jsimg` or `.jsimw` file) instance.

This script requires in input the filename and desired model name, and returns a single output, e.g.,

```
Network sn = JMT2LINE.load("examples/data/JMT/jmt_example.jsimg", "Mod1");
```

where `sn` is an instance of the class.

3.4.1 Supported JMT features

Table 3.4.1 lists the JSIMgraph/JSIMwiz model features supported by the `JMT2LINE` transformation. We indicate as "Fully" supported a feature that is supported in the import and such that the resulting model can be solved in LINE using at least `SolverJMT`. A feature with "Partial" support implies that some core aspects of this feature available in JSIM are not available in LINE.

A few notes are needed to clarify the entries with partial support:

- `Fork` and `Join` are supported with their default policies. Advanced policies, such as partial joins or setting a distribution for the forked tasks on each output link, are not supported yet.
- a single `Sink` and a single `Source` can be instantiated in a LINE model, whereas there is no such constraint in JMT.

Table 3.6: Supported JSIM features for automated model import and analysis

JMT Feature	Support	Notes
Distributions	Full	Phase-Type, Burst (MAP), Burst (MMPP2), Deterministic, Disabled, Exponential, Erlang, Gamma, Hyperexponential, Coxian, Logistic, Pareto, Uniform, Zero Service Time, Replayer, Weibull
Classes	Full	Open class, Closed class, Class priorities
Metrics	Full	Number of customers, Residence Time, Throughput, Response Time, Throughput per sink, Utilization, Arrival Rate

Continued on next page

Table 3.6 – *Continued from previous page*

JMT Feature	Support	Notes
Nodes	Full	Finite Capacity Region, ClassSwitch, Place, Delay, Logger, Queue, Router, Transition
Routing	Full	Random, Probabilities, Round Robin, Join the Shortest Queue
Mechanisms	Full	Polling, Switchover times
Scheduling	Full	FCFS, HOL, LCFS, LCFS-PR, SIRO (Random), SJF, SEPT, LJF, LEPT, PS, DPS, GPS, PS Priority, DPS Priority, GPS Priority
Nodes	Partial	Fork, Join, Source, Sink
Distributions	No	Burst (General), Normal
Nodes	No	Scaler, Semaphore
Routing	No	Shortest Response Time, Least Utilization, Fastest Service, Load Dependent, Class Switch Routing
Metrics	No	Drop rate, Response time per sink, Power
Scheduling	No	LPS, EDD, EDF, TBS, SRPT, QBPS
Mechanisms	No	Load Dependence, Retrial, Impatience, Soft deadlines, Parallelism, Heterogeneous servers, Server Compatibilities, Setup times, Polling, Switchover times

Chapter 4

Analysis methods

4.1 Performance metrics

As discussed earlier, LINE supports a set of steady-state and transient performance metrics. Table 4.1 summarizes the definition of the associated random variables. For each metric, one or more analysis types may be available, which are extensively discussed in the next sections.

Table 4.1: Performance metrics

Metric	Acronym	Description
Queue-length	`QLen`	Number of jobs of class r (or chain-c) residing at a node i
Utilization	`Util`	Utilization of class-r (or chain-c) jobs at node i, scaled in [0,1] for multi-server nodes, equal to `QLen` at infinite server nodes
Response time	`RespT`	Time that a class-r (or chain-c) jobs spends for a single visit at node i
Residence time	`ResidT`	Cumulative time that a class-r (or chain-c) jobs spends across all visits at node i
Arrival rate	`ArvR`	Arrival rate of class-r (or chain-c) jobs at node i
Throughput	`Tput`	Throughput of class-r (or chain-c) jobs at node i
System Response time	`SysRespT`	For an open chain c, this is the time from leaving the source to arriving at the sink for *any* class in the chain. For a closed chain c, this is the interval of time between two successive visits to the reference station in any two *completing classes* within the chain.
System Throughput	`SysTput`	For an open chain c, this is the departure rate towards the sink for *any* class in the chain. For a closed chain c, this is the rate of arrival of *completing classes* in the chain at the reference station.

4.2 Steady-state analysis

4.2.1 Station average performance

LINE decouples network specification from its solution, allowing to evaluate the same model with multiple solvers. Model analysis is carried out in LINE according to the following general steps:

Step 1: Definition of the model. This proceeds as explained in the previous chapters.

Step 2: Instantiation of the solver(s). A solver is an instance of the `Solver` class. LINE offers multiple solvers, which can be configured through a set of common and individual solver options. For example,

```
val solver: SolverJMT = SolverJMT(model)
```

returns a handle to a simulation-based solver based on JMT, configured with default options.

Step 3: Solution. Finally, this step solves the network and retrieves the concrete values for the performance indexes of interest. This may be done as follows, e.g.,

```
// QN(i,r): mean queue-length of class r at station i
val QN: Matrix = solver.avgQLen()
// UN(i,r): utilization of class r at station i
val UN: Matrix = solver.avgUtil()
// RN(i,r): mean response time of class r at station i (per visit)
val RN: Matrix = solver.avgRespT()
// TN(i,r): mean throughput of class r at station i
val TN: Matrix = solver.avgTput()
// AN(i,r): mean arrival rate of class r at station i
val AN: Matrix = solver.avgArvR()
// WN(i,r): mean residence time of class r at station i (summed on visits)
val WN: Matrix = solver.avgResidT()
```

Alternatively, all the above metrics may be obtained in a single method call as

```
SolverResults results = solver.avg();
Matrix QN = results.QN;
Matrix UN = results.UN;
Matrix RN = results.RN;
Matrix TN = results.TN;
Matrix AN = results.AN;
Matrix WN = results.WN;
```

In the methods above, LINE assigns station and class indexes (e.g., i, r) in order of creation in order of creation of the corresponding station and class objects. However, large models may be easier to debug by checking results using class and station names, as opposed to indexes. This can be done either by requesting LINE to build a table with the result

```
val avgTable: AvgTable = solver.avgTable()
```

which however tends to be a rather slow data structure to use in case of repeated invocations of the solver, or by indexing the matrices returned by `avg` using the model objects. That is, if the first instantiated node is `queue` with name `MyQueue` and the second instantiated class is `cclass` with name `MyClass`, then the following commands are equivalent

```
QN.get(0, 1); // 0-based indexing
QN.get(queue, cclass); // object-based indexing
QN.get(model.getStationIndex("MyQueue"), model.getClassIndex("MyClass"));
```

Similar methods are defined to obtain aggregate performance metrics at chain level at each station, namely `avgQLenChain` for queue-lengths, `avgUtilChain` for utilizations, `avgRespTChain` for response times, `avgTputChain` for throughputs, and the `avgChain` method to obtain all the previous metrics.

4.2.2 Station response time distribution

`SolverFluid` supports the computation of response time distributions for individual classes through the `getCdfRespT` function. The function returns the response time distribution for every station and class.

```
SolverFluid solver = new SolverFluid(model);
DistributionResult FC = solver.cdfRespT();
```

4.2.3 System average performance

LINE also allows users to analyze models for end-to-end performance indexes such a system throughput or system response time. However, in models with class switching the notion of system-wide metrics can be ambiguous. For example, consider a job that enters the network in one class and departs the network in another class. In this situation one may attribute system response time to either the arriving class or the departing one, or attempt to partition it proportionally to the time spent by the job within each class. In general, the right semantics depends on the aim of the study.

LINE tackles this issue by supporting only the computation of system performance indexes *by chain*, instead than by class. In this way, since a job switching from a class to another remains by definition in the same chain, there is no ambiguity in attributing the system metrics to the chain. The solver functions `avgSys` and `avgSysTable` return system response time and system throughput per chain as observed: (i) upon arrival to the sink, for open classes; (ii) upon arrival to the reference station, for closed classes.

In some cases, it is possible that a chain visits multiple times the reference station before the job completes. This also affects the definition of the system averages, since one may want to avoid counting each visit as a completion of the visit to the system. In such cases, LINE allows users to specify which classes of the chain can complete at the reference station. For example, in the code below we require that a job visits reference station 1 twice, in classes 1 and 2, but completes at the reference station only when arriving in class 2. Therefore, the system response time will be counted between successive passages in class 2.

```
ClosedClass class1 = new ClosedClass(model, "ClosedClass1", 1, queue, 0);
```

```
ClosedClass class2 = new ClosedClass(model, "ClosedClass2", 0, queue, 0);

class1.setCompletes(false);

// 2-classes routing matrix
Matrix[][] P = new Matrix[2][2];
P[0][0] = Matrix.create(new double[][]{{0,1},{0,0}}); // routing within class 1
P[0][1] = Matrix.create(new double[][]{{0,0},{1,0}}); // routing from class 1 to class 2
P[1][0] = Matrix.create(new double[][]{{0,0},{1,0}}); // routing within class 2
P[1][1] = Matrix.create(new double[][]{{0,1},{0,0}}); // routing from class 2 to class 2

model.link(P);
```

Note that the `completes` property of a class always refers to the reference station for the chain.

4.3 Specifying states

In some analyses it is important to specify the state of the network, for example to assign the initial position of the jobs in a transient analysis. We thus discuss the support in LINE for state modeling.

4.3.1 Station states

We begin by explaining how to specify a state s_0 for a station. For example, it is not supported for shortest job first (`SchedStrategy.SJF`) scheduling, in which state must include the service time samples for the jobs and it is therefore a continuous quantity.

Suppose that the network has R classes and that service distributions are phase-type, i.e., that they inherit from `Markovian`. Let K_r be the number of phases for the service distribution in class r at a given station. Then, we define three types of state variables:

- c_j: class of the job waiting in position $j \leq b$ of the buffer, out of the b currently occupied positions. If $b = 0$, then the state vector is indicated with a single empty element $c_1 = 0$.
- k_j: service phase of the job waiting in position $j \leq b$ of the buffer, out of the b currently occupied positions.
- n_r: total number of jobs of class r in the station
- b_r: total number of jobs of class r in the station's buffer
- s_{rk}: total number of jobs of class r running in phase k in the server

Here, by phase we mean the number of states of a distribution of class `Markovian`. If the distribution is not Markovian, then there is a single phase. With these definitions, the table below illustrates how to specify in LINE a valid state for a station depending on its scheduling strategy. There, S is the number of servers of the queueing station. All state variables are non-negative integers. The `SchedStrategy.EXT` policy is

Table 4.2: State descriptors for Markovian scheduling policies

Sched. strategy	Station state vector	State condition
EXT	$[Inf, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_k s_{rk} = 1, \forall r$
FCFS, HOL, LCFS	$[c_b, ..., c_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
LCFSPR	$[c_b, k_b, ..., c_1, k_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
SEPT, SIRO	$[b_1, ..., b_R, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
PS, DPS, GPS, INF	$[s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	None

used for the `Source` node, which may be seen as a special station with an infinite pool of jobs sitting in the buffer and a dedicated server for each class $r = 1, ..., R$.

States can be manually specified or enumerated automatically. LINE library functions for handling and generating states are as follows:

- `State.fromMarginal`: enumerates all states that have the same marginal state $[n_1, n_2, ..., n_R]$.
- `State.fromMarginalAndRunning`: restricts the output of `State.fromMarginal` to states with given number of running jobs, irrespectively of the service phase in which they currently run.
- `State.fromMarginalAndStarted`: restricts the output of `State.fromMarginal` to states with given number of running jobs, all assumed to be in service phase $k = 1$.
- `State.fromMarginalBounds`: similar to `State.fromMarginal`, but produces valid states between given minimum and maximum value of the number of resident jobs.
- `State.toMarginal`: extracts marginal statistics from a state, such as the total number of jobs in a given class that are running at the station in a certain phase.

Note that if a function call returns an empty state (`[]`), this should be interpreted as an indication that no valid state exists that meets the required criteria. Often, this is because the state supplied in input is invalid.

Example

We consider the example network in . We look at the state of station 3, which is a multi-server FCFS station. There are 4 classes all having exponential service times except class 2 that has Erlang-2 service times. We are interested to states with 2 running jobs in class 1 and 1 in class 2, and with 2 jobs, respectively of classes 3 and 4, waiting in the buffer. We can automatically generate this state space, which we store in the `space` variable, as:

```
val space = State.fromMarginalAndRunning(model, 2, intArrayOf(2,1,1,1), intArrayOf(2,1,0,0))
```

Here, each row of `space` corresponds to a valid state. The argument gives the number of jobs in the node for the 4 classes, while gives the number of running jobs in each class. This station has four valid states, differing on whether the class-2 job runs in the first or in the second phase of the Erlang-2 and on the relative position of the jobs of class 3 and 4 in the waiting buffer.

To obtain states where the jobs have just started running, we can instead use

```
val space = State.fromMarginalAndStarted(model, 2, intArrayOf(2,1,1,1), intArrayOf(2,1,0,0))
```

We see that the above state space restricted the one obtained with `State.fromMarginalAndRunning` to states where the job in class 1 is always in the first phase.

If we instead remove the specification of the running jobs, we can use `State.fromMarginal` to generate all possible combinations of states depending on the class and phase of the running jobs. In the example, this returns a space of 20 possible states.

```
val space = State.fromMarginal(model, 2, intArrayOf(2,1,1,1))
```

Assigning a prior to an initial state

It is possible to assign the initial state to a station using the `setState` function on that station's object. LINE offers the possibility to specify a prior probability on the initial states, so that if multiple states have a non-zero prior, then the solver will need to solve an independent model using each one of those initial states, and then carry out a weighting of the results according to the prior probabilities. The default is to assign a probability 1.0 to the *first* specified state. The functions `setStatePrior` and `getStatePrior` can be used to check and change the prior probabilities for the initial states specified for a station or stateful node.

4.3.2 Network states

A collection of states that are valid for each station is not necessarily valid for the network as a whole. For example, if the sum of jobs of a closed class exceeds the population of the class, then the network state would be invalid. To identify these situations, LINE requires to specify the initial state of a network using functions supplied by the `Network` class. These functions are `initFromMarginal`, `initFromMarginalAndRunning`, and `initFromMarginalAndStarted`. They require a matrix `n` with elements (i, r) specifying the total number of resident class-r jobs at node i and the latter two require a matrix `s` with elements (i, r) with the number of running (or started) class-r jobs at node i. The user can also manually verify if the supplied network state is going to be valid using `State.isValid`.

It is also possible to request LINE to automatically identify a valid initial state, which is done using the `initDefault` function available in the `Network` class. This is going to select a state where:

- no jobs in open classes are present in the network;
- jobs in closed classes all start at their reference stations;

- the servers at each reference station are occupied by jobs of in class order, i.e., jobs in the firstly created class are assigned to the server, then spare server are allocated to jobs in the second class, and so forth;
- service or arrival processes are initialized in phase 1 for each job;
- if the scheduling strategy requires it, jobs are ordered in the buffer by class, with the firstly created class at the head and the lastly created class at the tail of the buffer.

The `initFromAvgQLen` method is a wrapper for `initFromMarginal` to initialize the system as close as possible to the average steady-state distribution of the network. Since averages are typically not integer-valued, this function rounds the average values to the nearest integer and adjusts the result to ensure feasibility of the initialization.

4.3.3 Initialization of transient classes

Because of class-switching, it is possible that a class r with a non-empty population at time $t = 0$ becomes empty at some positive time $t' > t$ without ever being visited again by any job. LINE allows users to place jobs in transient classes and therefore it will not trigger an error in the presence of this situation. If a user wishes to prohibit the use of a class at a station, it is sufficient to specify that the corresponding service process uses the `Disabled` distribution.

Certain solvers may incur problems in identifying that a class is transient and in setting to zero its steady-state measures. For example, the `JMT` solver uses a heuristic whereby a class is considered transient if it has fewer samples than jobs initially placed in the corresponding chain the class belongs to. For such classes, `JMT` will set the values of steady-state performance indexes to zero.

4.3.4 State space generation

As discussed in Example 3, the state space of a model can be obtained by either invoking `model.stateSpace()` or `solver.stateSpace()` on an instant of the CTMC solver, where the latter returns the state space cached during the solution of the CTMC.

LINE supports two state space generation methods, configurable using the option `options.config.state_space_gen` of the CTMC solver. Details may be found in Table 5.2.2.

4.4 Transient analysis

So far, we have seen how to compute steady-state average performance indexes, which are given by

$$E[n] = \lim_{t \to +\infty} E[n(t)]$$

where $n(t)$ is an arbitrary performance index, e.g., the queue-length of a given class at time t.

We now consider instead the computation of the quantity $E[n(t)|s_0]$, which is the *transient average* of the performance index, conditional on a given initial system state s_0. Compared to $n(t)$, this quantity averages the system state at time t across all possible evolutions of the system from state s_0 during the t time units, weighted by their probability. In other words, we observe all possible stochastic evolutions of the system from state s_0 for t time units, recording the final values of $n(t)$ in each trajectory, and finally average the recorded values at time t to obtain $E[n(t)|s_0]$.

4.4.1 Computing transient averages

The computation of transient metrics proceeds similarly to the steady-state case. We first obtain the handles for transient averages:

```
Network model = Gallery.gallery_cqn(2); // closed single class queueing network
TranHandles handles = model.getTranHandles();
Matrix Qt = handles.Qt;
Matrix Ut = handles.Ut;
Matrix Tt = handles.Tt;
```

After solving the model, we will be able to retrieve *both* steady-state and transient averages as follows

```
TranResults results = new SolverCTMC(model, "timespan", new Matrix("[0,1]")).tranAvg(Qt, ...
Ut, Tt);
Matrix QNt = results.QNt;
Matrix UNt = results.UNt;
Matrix TNt = results.TNt;
// PlotUtils.plot(QNt.getCol(0), QNt.getCol(1));
```

The transient average queue-length at node i for class r is stored within `QNt` in row i and column r.

Note that the above code specifies a maximum time t for the output time series. This can be done using the `timespan` solver option. This applies also to average metrics. In the following example, the first model is solved at steady-state, while the second model reports averages at time $t = 1$ after initialization

```
// Steady-state analysis
new SolverCTMC(model).getAvgTable().print();

// Transient analysis at t=1
SolverOptions options = new SolverOptions();
options.timespan = new double[]{0, 1};
new SolverCTMC(model, options).getAvgTable().print();
```

4.4.2 First passage times into stations

When the model is in a transient, the average state seen upon arrival to a station changes over time. That is, in a transient, successive visits by a job may experience different response time distributions. The function `getTranCdfRespT`, implemented by `SolverJMT` offers the possibility to obtain this distribution given the

initial state specified for the model. As time passes, this distribution will converge to the steady-state one computed by solvers equipped with the function `getCdfRespT`.

However, in some cases one prefers to replace the notion of response time distribution in transient by the one of *first passage time*, i.e., the distribution of the time to complete the *first visit* to the station under consideration. The function `getTranCdfFirstPassT` provides this distribution, assuming as initial state the one specified for the model, e.g., using `setState` or `initDefault`. This function is available only in `SolverFluid` and has a similar syntax as `getCdfRespT`.

4.5 Sample path analysis

With LINE is also possible to obtain a particular sample path from the stochastic process underlying the queueing network. The following functions are available for this purpose:

- `sample`: returns a data structure including the time-varying state of a given stateful node, labelled with information about the events that changed the node state.
- `sampleAggr`: returns a data structure similar to the one provided by `sample`, but where the state is aggregate to count the number of jobs in each class at the node.
- `sampleSys`: similar to the `sample` function, but returns the state of every stateful node in the model.
- `sampleSysAggr`: similar to the `sampleAggr` function, but returns the aggregated state of every stateful node in the model.

It is worth noting that the `JMT` solver only supports `sampleAggr` since the simulator does not offer a simple way to extra detailed data such as phase change information in the service process. This information is instead available with the `SSA` solver.

For example, the following command extract a sample path consisting of 10 samples for a $APH(2)/M/1$ queue:

```
Network model = Gallery.gallery_aphm1();
SamplePath samplePath = new SolverJMT(model).sampleAggr(model.getNodes().get(1), 10);
System.out.println(Arrays.toString(samplePath.getT()));
System.out.println(Arrays.toString(samplePath.getState()));
```

In the example, refers to the time since initialization at which the node 2 (here the $APH(2)/M/1$ queueing station) enters the state shown in the second column.

If we repeat the same experiment with the `SSA` solver and using the `sampleSys` function, we now have the full state space of the model, including both the source and the queueing station:

```
Network model = Gallery.gallery_aphm1();
SamplePath samplePath = new SolverSSA(model).sampleSys(10);
System.out.println(Arrays.toString(samplePath.getT()));
for (int i = 0; i < samplePath.getState().size(); i++) {
```

```
        System.out.println("State " + i + ": " + Arrays.toString(samplePath.getState().get(i)));
    }
```

4.6 Sensitivity analysis and numerical optimization

Frequently, performance and reliability analysis requires to change one or more model parameters to see the sensitivity of the results or to optimize some goal function. In order to do this efficiently, we have discussed before the internal representation of the `Network` objects used within the LINE solvers. By applying changes directly to this internal representation it is possible to considerably speed-up the sequential evaluation of models as discussed next.

4.6.1 Fast parameter update

Successive invocations of `getStruct()` will return a cached copy of the `NetworkStruct` representation, unless the user has called `model.refreshStruct()` or `model.reset()` in-between the invocations. The `refreshStruct` function regenerates the internal representation, while `reset` destroys it, together with all other representations and cached results stored in the `Network` object. In the case of `reset`, the internal data structure will be regenerated at the next `refreshStruct()` or `getStruct()` call.

The performance cost of updating the representation can be significant, as some of the structure array field require a dedicated algorithm to compute. For example, finding the chains in the model requires an analysis of the weakly connected components of the network routing matrix. For this reason, the `Network` class provides several functions to selectively refresh only part of the `NetworkStruct` representation, once the modification has been applied to the objects (e.g., stations, classes, ...) used to define the network. These functions are as follows:

- `refreshArrival`: this function should be called after updating the inter-arrival distribution at a `Source`.
- `refreshCapacity`: this function should be called after changing buffer capacities, as it updates the `capacity` and `classcapacity` fields.
- `refreshChains`: this function should be used after changing the routing topology, as it refreshes the `rt`, `chains`, `nchains`, `nchainjobs`, and `visits` fields.
- `refreshPriorities`: this function updates class priorities in the `classprio` field.
- `refreshScheduling`: updates the `sched`, and `schedparam` fields.
- `refreshProcesses`: updates the `mu`, `phi`, `phases`, `rates` and `scv` fields.

For example, suppose we wish to update the service time distribution for class-1 at node 1 to be exponential with unit rate. This can be done efficiently as follows:

```
queue.setService(class1, new Exp(1.0));
model.refreshService();
```

4.6.2 Refreshing a network topology with non-probabilistic routing

The `resetNetwork` function should be used before changing a network topology with non-probabilistic routing. It will destroy by default all class switching nodes. This can be avoided if the function is called as, e.g., `model.resetNetwork(false)`. The default behavior is though shown in the next example

```
Network model = new Network("model");
ClassSwitch node1 = new ClassSwitch(model, "CSNode", Matrix.create(new ...
double[][]{{0,1},{0,1}}));
Queue node2 = new Queue(model, "Queue1", SchedStrategy.FCFS);
System.out.println("Before reset: " + model.getNodes().size());
\texttt{List\textlangle Node\textrangle} remaining = model.resetNetwork();
System.out.println("After reset: " + remaining.size());
```

As shown, `resetNetwork` updates the station indexes and the revised list of nodes that compose the topology is obtained as a return parameter. To avoid stations to change index, one may simply create `ClassSwitch` nodes as last before solving the model. This node list can be employed as usual to reinstantiate new stations or `ClassSwitch` nodes. The `addLink`, `setRouting`, and possibly the `setProbRouting` functions will also need to be re-applied as described in the previous sections.

4.6.3 Saving a network object before a change

The object, and its inner objects that describe the network elements, are always passed by reference. The `copy` function should be used to clone LINE objects, for example before modifying a parameter for a sensitivity analysis. This function recursively clones all objects in the model, therefore creating an independent copy of the network. For example, consider the following code

```
Network modelByRef = model; modelByRef.setName("myModel1");
Network modelByCopy = model.copy(); modelByCopy.setName("myModel2");
```

Using the `getName` function it is then possible to verify that `model` has now name `myModel1`, since the first assignment was by reference. Conversely, `modelByCopy.setName` did not affect the original `model` since this is a clone of the original network.

Chapter 5

Network solvers

5.1 Overview

Solvers analyze objects of class to return average, transient, distributions, or state probability metrics. A solver can implement one or more *methods*, which although featuring a similar overall solution strategy, they can differ significantly from each other in the way this strategy is actually implemented and on whether the final solution is exact or approximate.

A 'method' flag can be passed upon invoking a solver to specify the solution method that should be used. For example, the following invocations are identical:

```
new SolverMVA(model, "exact").avgTable();
new SolverMVA(model, "method", "exact").avgTable();
SolverOptions opt = SolverMVA.defaultOptions(); opt.method = "exact";
new SolverMVA(model, opt).avgTable();
```

In what follows, we describe the general characteristics and supported model features for each solver available in LINE and their methods.

Available solvers

The following solvers are available within LINE 3.0.x:

- LINE: This solver uses an algorithm to select the best solution method for the model under consideration, among those offered by the other solvers. Analytical solvers are always preferred to simulation-based solvers. This solver is implemented by the LINE class.

- CTMC: This is a solver that returns the exact values of the performance metrics by explicit generation of the continuous-time Markov chain (CTMC) underpinning the model. As the CTMC typically incurs state-space explosion, this solver can successfully analyze only small models. The CTMC solver is the only method offered within LINE that can return an exact solution on all Markovian models, all other

solvers are either approximate or are simulators. This solver is implemented by the `SolverCTMC` class.

- `FLUID`: This solver analyzes the model by means of an approximate fluid model, leveraging a representation of the queueing network as a system of ordinary differential equations (ODEs). The fluid model is approximate, but if the servers are all PS or INF, it can be shown to become exact in the limit where the number of users and the number of servers in each node grow to infinity [36]. This solver is implemented by the `SolverFluid` class.

- `JMT`: This is a solver that uses a model-to-model transformation to export the LINE representation into a JMT simulation (JSIM) or analytical (JMVA) models [3]. The JSIM simulation solver can analyze also non-Markovian models, in particular those involving deterministic or Pareto distributions, or empirical traces. This solver is implemented by the `SolverJMT` class.

- `MAM`: This is a matrix-analytic method solver, which relies on quasi-birth death (QBD) processes to analyze open queueing systems. This solver is implemented by the `SolverMAM` class.

- `MVA`: This is a solver based on approximate and exact mean-value analysis. This solver is typically the fastest and offers very good accuracy in a number of situations, in particular models where stations have a single-server. This solver is implemented by the `SolverMVA` class.

- `NC`: This solver uses a combination of methods based on the normalizing constant of state probability to solve a model. The underpinning algorithm are particularly useful to compute marginal and joint state probabilities in queueing network models. This solver is implemented by the `SolverNC` class.

- `SSA`: This is a Stochastic Simulation Algorithms based on the CTMC representation of the model. Contrary to the `JMT` simulator, which has online estimators for all the performance metrics, `SSA` estimates only the probability distribution of the system states, indirectly deriving the metrics after the simulation is completed. Moreover, the `SSA` execution can more efficiently parallelized on multi-core machines. Moreover, it is possible to retrieve the evolution over time of each node state, including quantities that are not loggable in `JMT`, e.g., the active phase of a service or arrival distribution. This solver is implemented by the `SolverSSA` class.

5.2 Solution methods

We now describe the solution methods available within the solvers. Table 5.1 provides a global summary. Some of the listed methods (e.g., `mg1`) are not associated to a specific solver, as they do not fall in one of the reference formalisms. A solver that runs these methods can be instantiated as follows, e.g.:

```
NetworkSolver solver = LINE.load("mg1", model);
solver.getAvgTable().print();
```

Note that the `LINE.load` notation can also be used to instantiate a custom solver pre-configured with the specified method. For example

```
Solver solver = LINE.load("ctmc", model);
```

runs the `CTMC` solver with default options. Solver-specific methods can be specified by appending their name to the method option, e.g. this command creates the `CTMC` solver with `gpu` method enabled:

```
NetworkSolver solver = LINE.load("ctmc.gpu", model);
```

Table 5.1: Solution methods for `Network` solvers.

Solver	Method	Description	Refs.
`CTMC`	`default`	Solution based on global balance	[5, §2.1.2]
`FLUID`	`default`	ODE-based mean field approximations	[37,43]
`FLUID`	`matrix`	Alias for the `default` method	[37,43]
`FLUID`	`closing`	Fluid with closing method for open classes	[5][p. 507]
`FLUID`	`statedep`	Kurtz's mean field ODEs for closed models	[37]
`FLUID`	`softmin`	Smoothed `statedep` with *softmin* replacing *min* functions	–
`JMT`	`default`	Alias for the `jsim` method	–
`JMT`	`jmva`	Alias for the `jmva.mva` method	–
`JMT`	`jmva.mva`	Exact MVA in JMVA	[40]
`JMT`	`jmva.recal`	Exact RECAL algorithm in JMVA	[18]
`JMT`	`jmva.comom`	Exact CoMoM algorithm in JMVA	[8]
`JMT`	`jmva.amva`	Approximate MVA, alias for `jmva.bs`.	–
`JMT`	`jmva.aql`	AQL algorithm in JMVA	[49]
`JMT`	`jmva.bs`	Bard-Schweitzer algorithm in JMVA	[5, §9.1.1]
`JMT`	`jmva.chow`	Chow algorithm in JMVA	[16]
`JMT`	`jmva.dmlin`	De Souza-Muntz Linearizer in JMVA	[19]
`JMT`	`jmva.lin`	Linearizer algorithm in JMVA	[15]
`JMT`	`jmva.ls`	Logistic sampling in JMVA	[9]
`JMT`	`jsim`	Exact discrete-event simulation in JSIM	[3]
`MAM`	`default`	Matrix-analytic solution of structured QBDs	[29]
`MAM`	`dec.source`	Decomposition with arrivals as from the source	–
`MAM`	`dec.poisson`	Decomposition based on Poisson arrival flows	–
`MAM`	`dec.mna`	Decomposition based on MNA method	[33]
`MVA`	`default`	Approximate MVA, same as `qd` option	–
`MVA`	`amva`	Approximate MVA, same as `qd` option	–

Continued on next page

Table 5.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
MVA	bs	Bard-Schweitzer approximate MVA	[5, §9.1.1]
MVA	lin	Linearizer approximate MVA	[15]
MVA	qd	Queue-dependent approximate MVA	[13]
MVA	qdlin	Queue-dependent Linearizer approximate MVA	–
MVA	exact	Exact solution, method depends on model	–
MVA	mva	Alias for the `mva.amva` method	[40], [7]
MVA	aba.upper	Asymptotic bound analysis (upper bounds)	[5, §9.4]
MVA	aba.lower	Asymptotic bound analysis (lower bounds)	[5, §9.4]
MVA	bjb.upper	Balanced job bounds (upper bounds)	[12, Table 3]
MVA	bjb.lower	Balanced job bounds (lower bounds)	[12, Table 3]
MVA	gb.upper	Geometric square-root bounds (upper bounds)	[12]
MVA	gb.lower	Geometric square-root bounds (lower bounds)	[12]
MVA	pb.upper	Proportional bounds (upper bounds)	[12, Table3]
MVA	pb.lower	Proportional bounds (lower bounds)	[12, Table3]
MVA	sb.upper	Simple bounds (upper bounds, Thm. 3.2, $n = 3$)	[27, Table3]
MVA	sb.lower	Simple bounds (lower bounds, Eq. 1.6)	[27, Table3]
MVA	gig1.allen	Allen-Cunneen formula - GI/G/1	[5, §6.3.4]
MVA	gig1.heyman	Heyman formula - GI/G/1	–
MVA	gig1.kingman	Kingman upper bound- GI/G/1	[5, §6.3.6]
MVA	gig1.klb	Kramer-Langenbach-Belz formula - GI/G/1	[5, §6.3.4]
MVA	gig1.kobayashi	Kobayashi diffusion approximation - GI/G/1	[5, §10.1.1]
MVA	gig1.marchal	Marchal formula - GI/G/1	[5, §10.1.3]
MVA	gigk	Kingman approximation - GI/G/k	
MVA	mg1	Pollaczek–Khinchine formula - M/G/1	[5, §3.3.1]
MVA	mm1	Exact formula - M/M/1	[5, §6.2.1]
MVA	mmk	Exact formula - M/M/k (Erlang-C)	
NC	default	Alias for the `adaptive` method	–
NC	adaptive	Automated choice of deterministic method	–
NC	exact	Automated choice of exact solution method.	–
NC	ca	Multiclass convolution algorithm (exact)	–
NC	comom	Class-oriented method of moments for hommogeneous models (exact)	[8]
NC	cub	Grundmann-Moeller cubature rules	[9]
NC	mva	Product of throughputs on MVA lattice (exact)	[39, Eq. (47)]
NC	imci	Improved Monte carlo integration sampler	[47]
NC	kt	Knessl-Tier asymptotic expansion	[30]
NC	le	Logistic asymptotic expansion	[9]

Continued on next page

Table 5.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
NC	ls	Logistic sampling	[9]
NC	nr.logit	Norlund-Rice integral with logit transformation	[11]
NC	nr.probit	Norlund-Rice integral with probit transformation	[11]
NC	panacea	Panacea asymptotic expansion	[35], [41]
NC	rd	Reduction heuristic	[11]
NC	sampling	Automated selection of sampling method	–
SSA	default	Alias for nrm if the model supports it, otherwise serial.	–
SSA	nrm	Next reaction method for population models (e.g., PS/INF scheduling).	[1]
SSA	serial	CTMC stochastic simulation on a single core	[26]
SSA	para	Parallel simulations (independent replicas)	–

5.2.1 LINE

The LINE class, also callable with the alias `SolverAuto`, provides interfaces to the core solution functions (e.g., `avg`, ...) that dynamically bind to one of the other solvers implemented in LINE (`CTMC`, `NC`, ...). It is often difficult to identify the best solver without some performance results on the model, for example to determine if it operates in light, moderate, or heavy-load regime.

Therefore, heuristics are used to identify a solver based on structural properties of the model, such as based on the scheduling strategies used at the stations as well as the number of jobs, chains, and classes. Such heuristics, though, are independent of the core function called, thus it is possible that the optimal solver does not support the specific function called (e.g., `getTranAvg`). In such cases the LINE solver determines what other solvers would be feasible and prioritizes them in execution time order, with the fastest one on average having the higher priority. Eventually, the solver will be always able to identify a solution strategy, through at least simulation-based solvers such as `JMT` or `SSA`.

5.2.2 CTMC

The `SolverCTMC` class solves the model by first generating the infinitesimal generator of the and then calling an appropriate solver. Steady-state analysis is carried out by solving the global balance equations defined by the infinitesimal generator. If the `keep` option is set to `true`, the solver will save the infinitesimal generator in a temporary file and its location will be shown to the user.

Transient analysis is carried out by numerically solving Kolmogorov's forward equations using MATLAB's ODE solvers. The range of integration is controlled by the `timespan` option. The ODE solver

choice is the same as for `SolverFluid`.

The CTMC solver heuristically limits the solution to models with no more than 6000 states. The `force` option needs to be set to `true` to bypass this control. In models with infinite states, such as networks with open classes, the `cutoff` option should be used to reduce the CTMC to a finite process. If specified as a scalar value, `cutoff` is the maximum number of jobs that a class can place at an arbitrary station. More generally, a matrix assignment of `cutoff` indicates to LINE that `cutoff` has in row i and column r the maximum number of jobs of class r that can be placed at station i.

Details on the additional configuration options of the CTMC solver is given in the next table.

Table 5.2: `SolverCTMC` configuration options (Kotlin)

Option	Value	Description
`options.config.hide_immediate`	`Boolean`	If true, immediate transitions are hidden from the CTMC.
`options.config.state_space_gen`	`"reachable"`	Direct state space enumeration from initial state.
`options.config.state_space_gen`	`"full"`	Direct state space enumeration from all possible initial states.
`options.timestep`	`Double`	Fixed time interval for transient analysis. If specified, generates equally-spaced time points instead of adaptive stepping.

5.2.3 FLUID

This solver is based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [37]. The latter is based on Kurtz's mean-field approximation theory. The fluid ODEs are normally solved with a Java port of the LSODA algorithm for stiff and non-stiff ordinary differential equations. More details about the port are available at: https://github.com/imperial-qore/lsoda-java.

ODE variables corresponding to an infinite number of jobs, as in the job pool of a source station, or to jobs in a disabled class are not included in the solution vector. These rules apply also to the `options.init_sol` vector.

The solution of models with FCFS stations maps these stations into corresponding PS stations where the service rates across classes are set identical to each other with a service distribution given by a mixture of the service processes of the service classes. The mixture weights are determined iteratively by solving a sequence of PS models until convergence. Upon initializing FCFS queues, jobs in the buffer are all initialized in the first phase of the service.

5.2.4 JMT

The class is a wrapper for the `JMT` and consists of a model-to-model transformation from the data structure into the JMT's input XML formats (either `.jsimg` or `.jmva`) and a corresponding parser for JMT's results. Upon first invocation, the `JMT` JAR archive will be searched in the MATLAB path and if unavailable automatically downloaded.

This solver offers two main methods. The default method is the JSIM solver (method), which runs JMT's discrete-event simulator. For parallel simulation, the solver supports both serial and parallel execution methods. The alternative method is the JMVA analytical solver (method), which is applicable only to queueing network models that admit a product-form solution. This can be verified calling `model.hasProductFormSolution` prior to running the JMVA solver.

In the transformation to JSIM, artificial nodes will be automatically added to the routing table to represent class-switching nodes used in the simulator to specify the switching rules. One such class-switching node is defined for every ordered pair of stations (i, j) such that jobs change class in transit from i to j.

5.2.5 MAM

This is a basic solver for some Markovian open queueing systems that can be analyzed using matrix analytic methods. The core solver is based on the BU tools library for matrix-analytic methods [29]. The solution of open queueuing networks is based on traffic decomposition methods that compute the arrival process at each queue resulting from the superposition of multiple source streams.

5.2.6 MVA

The solver offers approximate mean value analysis (AMVA) (`options.method=`), but also exact MVA algorithms (`options.method=`). The default AMVA solver is based on Linearizer [15], unless there are two or less jobs in total within closed classes, in which case the solver runs the Bard-Schweitzer algorithm [44]. Extended queueing models are handled as follows:

- Non-exponential service times in FCFS nodes are handled only in the single-server case via the method selected in the `options.config.highvar` setting. By default high variance is ignored, as the FCFS solver tends to produce good result in closed models also without specialized corrections. It is alternatively possible to handle high variance either using the Diffusion-M/G/k interpolation from [10], casted with weights $a_i = b_i = 10^{-8}$, or using the high-variance MVA (HV-MVA) corrections proposed in [6, 38]. The multi-server extension is ongoing; we point to the NC solver for a version already available.
- Multi-servers are dealt with using the methods listed in Table 5.3 for the `options.config.multiserver` option. These are coupled with a modification of the Rolia-Sevcik correction [42], where in light-load the Rolia-Sevcik correction is treated as if there was a single server.
- Non-preemptive are dealt with using the methods listed in Table 5.3 for the configuration option `options.config.np_priority`. The solver feature in particular AMVA-CL and the shadow server methods [21].
- DPS queues are analyzed with a standard method similar to the biased processor sharing approximation reported in [32, §11.4]. Here, an arriving job of class r sees a queue-length in class $s \neq r$ scaled by the correction factor w_s/w_r, where w_s is the weight of class s.

- Limited load-dependence (intended here as other than multi-server) and class-dependence are handled through the correction factors proposed in [13]. If a station is both limited load-dependent and multi-server, then if the `softmin` method is chosen the solver will suitably combine the softmin term and the limited load-dependent correcting factors. Moreover, iterative queue-length corrections such as those applied by the AQL and Linearizer methods are also applied to these terms. Limited load-dependence (queue-dependent AMVA or QD-AMVA) is handled through correction factors.

- Fork-join networks are assumed to feature a direct acyclic graph (DAG) in-between forks and joins. They are analyzed by iteratively transforming the sibling tasks into jobs belonging to independent classes, using the algorithm specified in `options.config.fork_join`. If a fork has fan out f (i.e., the fork out-degree), in the implementation of the Heidelberger-Trivedi [28] method, one artificial open class is created for each of $f-1$ sibling task, while also retaining a task in the original class. The residence times along a branch are then treated as exponential random variables and their maximum, corresponding to the response time of the fork-join section, is computed using specialized results for this distribution. LINE supports this method, but uses as a default a custom variant whereby in which the original and artificial classes can take with probability $1/f$ any of the outgoing branches. While the latter can result in states that do not exist in the original model, since two sibling tasks may take the same branch, it is correct in expectation and it does not treat differently the artificial classes than the original class, which can be beneficial when the original class is *closed* and thus differs significantly from an *open* artificial class.

Solver-specific configuration options are reported in Table 5.3.

Table 5.3: `SolverMVA` configuration options (Kotlin)

Option	Value	Description
`options.config.multiserver`	`"default"`	Equals `"softmin"` at PS queues and `"seidmann"` at FCFS queues.
`options.config.multiserver`	`"seidmann"`	Seidmann's decomposition [45].
`options.config.multiserver`	`"softmin"`	QD-AMVA's softmin approximation [13].
`options.config.np_priority`	`"default"`	Non-preemptive priority handling. Equals `"cl"`.
`options.config.np_priority`	`"cl"`	Chandy-Lakshmi [21].
`options.config.np_priority`	`"shadow"`	Sevcik's shadow server [46].
`options.config.highvar`	`"default"`	Ignored - no correction applied.
`options.config.highvar`	`"interp"`	Diffusion-M/G/k interpolation from [10].
`options.config.highvar`	`"hvmva"`	High-variance MVA as in [6], extended to multiclass as [22, Eq. 3.21].
`options.config.fork_join`	`"default"`	Equals `"mmt"`.
`options.config.fork_join`	`"mmt"`	Mixed-model transformation [20]
`options.config.fork_join`	`"ht"`	Heidelberger-Trivedi [28]

5.2.7 NC

The `SolverNC` class implements a family of solution algorithms based on the normalizing constant of state probability of product-form queueing networks. Contrary to the other solvers, this method typicallly maps the problem to certain multidimensional integrals, allowing the use of numerical methods such as MonteCarlo sampling and asymptotic expansions in their approximation.

5.2.8 SSA

The `SolverSSA` class is a basic stochastic simulator for continuous-time Markov chains. It reuses some of the methods that underpin `SolverCTMC` to generate the network state space and subsequently simulates the state dynamics by probabilistically choosing one among the possible events that can incur in the system, according to the state spaces of each of node in the network. For efficiency reasons, states are tracked at the level of individual stations and, in some of the algorithms, hashed.

The state space is not generated upfront, but typically stored during the simulation, starting from the initial state. If the initialization of a station generates multiple possible initial states, `SSA` initializes the model using the first state found. The list of initial states for each station can be obtained using the `getInitState` functions of the `Network` class.

The `SSA` solver offers two comprehensive methods: `'serial'` and `'para'` (default). The serial method runs on a single core, while the parallel methods run on multicore

`SSA` solver also implements the much faster next reaction method (`'nrm'`, see [1]). However, this is available only for open and closed queueing models with specific scheduling disciplines, in particular `INF` and `PS`. The `'nrm'` method is default on such models. Moreover, by setting `options.config.state_space_gen` to `"full"` it is possible to explicitly generate during the simulation the simulated state space and its steady-state probability.

5.3 Supported language features and options

5.3.1 Solver features

Once a model is specified, it is possible to use the `getUsedLangFeatures` function to obtain a list of the features of a model. For example, the following conditional statement checks if the model contains a FCFS node

```
if (model.getUsedLangFeatures().list.SchedStrategy_FCFS) {
    // ...
}
```

Every LINE solver implements the `support` to check if it supports all language features used in a certain model

```
System.out.println(SolverJMT.supports(model));
```

5.3.2 Class functions

The table below lists the steady-state and transient analysis functions implemented by the solvers. Since the features of the LINE solver are the union of the features of the other solvers, in what follows it will be omitted from the description.

Table 5.4: Solver support for average performance metrics

		Network Solver						
Function	**Regime**	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
avg	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNode	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgSys	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgSysTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgArvR	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgArvRChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeArvRChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgQLen	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgQLenChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeQLenChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgRespT	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgRespTChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeRespTChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgSysRespT	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgTput	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgTputChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeTputChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgSysTput	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgUtil	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgUtilChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
avgNodeUtilChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
getTranAvg	Transient	✓	✓	✓				

The functions listed above with the Table suffix (e.g., avgTable) provide results in tabular format corresponding to the corresponding core function (e.g., avg). The features of the core functions are as follows:

- avg: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for each station and class.

Table 5.5: Solver support for advanced metrics

		Network Solver						
Function	**Regime**	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
getCdfRespT	Steady-state	✓	✓	✓	✓			
getProb	Steady-state	✓						
getProbAggr	Steady-state	✓	✓	✓		✓	✓	
getProbSys	Steady-state	✓						✓
getProbSysAggr	Steady-state	✓		✓			✓	
getProbNormConstAggr	Steady-state			✓		✓	✓	
getTranCdfPassT	Transient		✓					
getTranCdfRespT	Transient			✓				
getTranProb	Transient	✓						
getTranProbAggr	Transient	✓						
getTranProbSys	Transient	✓						
getTranProbSysAggr	Transient	✓						
sample	Transient							✓
sampleAggr	Transient			✓				✓
sampleSys	Transient							✓
sampleSysAggr	Transient			✓				✓

- avgChain: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for every station and chain.
- avgSys: returns the system response time and system throughput, as seen as the reference node, by chain.
- getCdfRespT: returns the distribution of response times (for one visit) for the stations at steady-state.
- avgNode: behaves similarly to avg, but returns performance metrics for each node and class. For example, throughputs at the sinks can be obtained with this method.
- getProb: returns state probabilities at equilibrium at a given station.
- getProbAggr: returns marginal state probabilities for jobs of different classes at a given station.
- getProbSys: returns joint probabilities for a given system state.
- getProbSysAggr: returns joint probabilities for jobs of different classes at all stations.
- getProbNormConstAggr: returns the normalizing constant of the state probabilities for the model.

- `getTranAvg`: returns transient mean queue length, utilization and throughput for every station and chain from a given initial state.
- `getTranCdfPassT`: returns the distribution of first passage times in transient regime.
- `getTranCdfRespT`: returns the distribution of response times in transient regime.
- `sample`: returns the transient marginal state for a station from a given initial state.
- `sampleAggr`: returns the transient marginal state for jobs of different classes at a given station from a given initial state.
- `sampleSys`: returns the transient marginal system state for a station from a given initial state.
- `sampleSysAggr`: returns the transient marginal system state for jobs of different classes at a given station from a given initial state.

5.3.3 Node types

The table below shows the node types supported by the different solvers. It should be noted that the `FLUID` solver is capable of handling and nodes, but due to low accuracy when run on open models this feature is disabled in the current release.

Table 5.6: Solver support for nodes

	Network Solver						
Strategy	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
Cache	✓				✓	✓	✓
ClassSwitch	✓	✓	✓	✓	✓	✓	✓
Delay	✓	✓	✓	✓	✓	✓	✓
Fork			✓		✓		
Join			✓		✓		
Queue	✓	✓	✓	✓	✓	✓	✓
Sink	✓		✓	✓	✓	✓	✓
Source	✓		✓	✓	✓	✓	✓

5.3.4 Scheduling strategies

The table below shows the supported scheduling strategies within LINE queueing stations. Each strategy belongs to a policy class:

- preemptive resume (`SchedStrategyType.PR`)

- non-preemptive (`SchedStrategyType.NP`)
- non-preemptive priority (`SchedStrategyType.NPPrio`).

The table primarily refeers to invocation of the `avg` methods. Specialized methods, such as transient or response time distribution analysis, may be available only for a subset of the scheduling strategies supported by a solver.

Table 5.7: Solver support for scheduling strategies

		Network Solver						
Strategy	**Class**	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
FCFS	NP	✓	✓	✓	✓	✓	✓	✓
INF	NP	✓	✓	✓	✓	✓	✓	✓
SIRO	NP	✓		✓		✓	✓	✓
SEPT	NP	✓		✓				✓
SJF	NP			✓				
HOL	NPPrio	✓		✓		✓		✓
PS	PR	✓	✓	✓	✓	✓	✓	✓
DPS	PR	✓	✓	✓		✓		✓
GPS	PR	✓		✓				✓
PSPRIO	PRPrio	✓		✓				✓
DPSPRIO	PRPrio	✓		✓				✓
GPSPRIO	PRPrio	✓		✓				✓

5.3.5 Statistical distributions

The table below summarizes the current level of support for arrival and service distributions within each solver. `Replayer` represents an empirical trace read from a file, which will be either replayed as-is by the JMT solver, or fitted automatically to a `Cox` by the other solvers. Note that JMT requires that the last row of the trace must be a number, *not* an empty row.

5.3.6 Solver options

Table 5.9 summarizes the main options available within the LINE solvers and their default values. Solver options are encoded in LINE in a structure array that is internally passed to the solution algorithms.

This can be specified as an argument to the constructor of the solver. For example, the following two constructor invocations are identical

```
Solver s = new SolverJMT(model);
SolverOptions opt = SolverJMT.defaultOptions(); s = new SolverJMT(model, opt);
```

Table 5.8: Solver support for statistical distributions

	Network Solver						
Distribution	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
APH	✓		✓	✓	✓	✓	✓
Coxian	✓	✓	✓	✓	✓	✓	✓
Exp	✓	✓	✓	✓	✓	✓	✓
Erlang	✓	✓	✓	✓	✓	✓	✓
HyperExp	✓	✓	✓	✓	✓	✓	✓
Disabled	✓	✓	✓	✓	✓	✓	✓
Det			✓				
Gamma			✓				
Lognormal			✓				
Pareto			✓				
Replayer			✓				
Uniform			✓				
Weibull			✓				

Modifiers to the default options can either be specified directly in the `options` data structure, or alternatively be specified as argument pairs to the constructor, i.e., the following two invocations are equivalent

```
Solver s = new SolverJMT(model, "samples", 1000000);
SolverOptions opt = SolverJMT.defaultOptions(); opt.samples = 1000000; s = new ...
SolverJMT(model, opt);
```

Available solver options are as follows:

- `cache` (`logical`) if set to `true` the solver after the first invocation will return the same result upon subsequent calls, without solving again the model. This option is `true` by default. Caching can be bypassed using the `refresh` methods (see Section 4.6).
- `config` (`struct`) this is data structure to pass solver-specific configuration options to customize the execution of particular methods.
- `cutoff` (`integer` ≥ 1) requires to ignore states where stations have more than the specified number of jobs. This is a mandatory option to analyze open classes using the CTMC solver.
- `force` (`logical`) requires the solver to proceed with analyzing the model. This bypasses checks and therefore can result in the solver either failing or requiring an excessive amount of resources from the system.
- `iter_max` (`integer` ≥ 1) controls the maximum number of iterations that a solver can use, where applicable. If `iter_max=` n, this option forces the `FLUID` solver to compute the ODEs over the

timespan $t \in [0, 10n/\mu^{\min}]$, where $\mu^{\min}$ is the slowest service rate in the model. For the `MVA` solver this option instead regulates the number of successive substitutions allowed in the fixed-point iteration.

- `iter_tol` (`double`) controls the numerical tolerance used to convergence of iterative methods. In the `FLUID` solver this option regulates both the absolute and relative tolerance of the ODE solver.
- `init_sol` (`solver dependent`) re-initializes iterative solvers with the given configuration of the solution variables. In the case of `MVA`, this is a matrix where element (i, j) is the mean queue-length at station i in class j. In the case of `FLUID`, this is a model-dependent vector with the values of all the variables used within the ODE system that underpins the fluid approximation.
- `keep` (`logical`) determines if the model-to-model transformations store on file their intermediate outputs. In particular, if `verbose`≥ 1 then the location of the `.jsimg` models sent to JMT will be printed on screen.
- `method` (`string`) configures the internal algorithm used to solve the model.
- `samples` (`integer` ≥ 1) controls the number of samples collected *for each* performance index by simulation-based solvers. `JMT` requires a minimum number of samples of $5 \cdot 10^3$ samples.
- `seed` (`integer` ≥ 1) controls the seed used by the pseudo-random number generators. For example, simulation-based solvers will give identical results across invocations only if called with the same `seed`.
- `stiff` (`logical`) requires the solver to use a stiff ODE solver.
- `timespan` (`real interval`) requires the transient solver to produce a solution in the specified temporal range. If the value is set to $double[]Inf, Inf$, where Inf is imported from `GlobalConstants`, the solver will only return a steady-state solution. For the `FLUID` solver and in simulation, this setting has the same computational cost of $double[]0.0, Inf$, therefore the latter is used as default for this solver.
- `timestep` (`double`) controls the fixed time interval for transient analysis in the `CTMC` solver. When specified, the solver generates equally-spaced time points instead of using adaptive time stepping. If not specified or set to empty, the solver uses adaptive ODE time stepping. This option only affects transient analysis and is ignored for steady-state computations.
- `tol` default numerical tolerance for all uses other than the ones where `iter_tol` is used.
- `verbose` controls the verbosity level of the solver. Supported levels are 0 for silent, 1 for standard verbosity, 2 for debugging.

Table 5.9: Default values of the LINE solver options and their default assignments (Kotlin)

Option	MVA	CTMC	FLUID	
cache	true	true	true	
config				
cutoff		(no default)		
force	false	false	false	
keep				
init_sol	null		null	
iter_max	10^3		10	
iter_tol	10^{-6}		10^{-4}	
method	"default"	"default"	"default"	
samples				
seed	Random	Random	Random	
stiff			true	
timespan		[Double.POSITIVE_INFINITY]	[0.0,Double.POSITIVE_INFINITY]	[0.0,Doub
tol		10^{-4}	10^{-4}	
verbose	1	1	1	

Chapter 6

Layered network models

In this chapter, we present the definition of the `LayeredNetwork` class, which encodes the support in LINE for a class of generalized layered stochastic networks. In their basic form, these models are called layered queueing networks (LQNs) and differ from regular queueing networks as servers, in order to process jobs, can issue synchronous and asynchronous calls among each others. We point to [23] and to the LQNS user manual for an introduction [24]. Contrary to the original LQNs, layered networks in LINE can also include non-queueing servers, such as caches, hence they may be conceptualized as more general layered stochastic networks.

The topology of call dependencies in a layered network makes it possible to partition the model into a set of layers, each consisting of a subset of the servers. Each of these layers is then solved in isolation, updating with an iterative procedure its parameters and performance metrics until the layers solutions jointly converge to a consistent solution.

6.1 Basics about layered networks

Layered network models describe a collection of resources called *tasks*, each representing for example a software server, that run on resources called *host processors*. Classes of service exposed by a task are called *entries*. Each entry is an endpoint at which a task can be invoked; for example, if a task represents a web server then its web pages may be described as different entries.

A special task, called the *reference task* is used to represent a group of system users. In this case, the host processor for a reference task can either be real, as in the case of users that are themselves software systems, or fictitious, as in the case of human users.

Each entry can be specified by a workflow of operations called *activities*, typically organized as a directed acyclic graph. The time demand that each activity places at the underpinning host processor is called its *host demand* and it is a random variable with a user-specified distribution.

Activity graphs may include *calls* to entries exposed by other tasks. This is an abstraction of the calls that distributed system components have among themselves. Calls can either be *synchronous*, *asynchronous*, or

forwarding. At present, LINE supports only the first two kinds of activities. Synchronous calls are requests that block the sender until a reply is received, while asynchronous calls are non-blocking and the sender execution can continue after issuing the call. Calls can either be repeated either *deterministic* or *stochastic*, meaning in the latter case that the number of calls issued is a random variable, e.g. geometrically distributed.

Contrary to ordinary layered queueing networks, a layered network in LINE can also feature *cache tasks*, *item entries*, and *cache-access* precedence relations.

- Cache tasks have the basic properties of tasks, but add three specific properties for caching: the total number of items, the cache capacity and the cache replacement policy. Cached items can be either contents or services. Cache capacity indicates the storage constraints of the cache.
- An item-entry provides instead access to a group of entries of a cache, Item-entries have the basic properties of entries, but add the property of the popularity of the items they give access to.
- A precedence relationship called *cache-access* is defined for the cache hit and miss activities under each item-entry. That is, it is possible to proceed to a different activity depending on whether the cache access produced a cache hit or cache miss. For example, a cache miss can produce a call to a remote entry to retrieve the missing content.

Note that the above extensions are not queueing-based and this explains why these models are referred to in LINE as layered networks and not as layered queueing networks. Similar to the latter, the analysis of a layered networks uses a decomposition of the model into a set of submodels, each being a object, which are then iterative analyzed using different solution methods.

6.2 LayeredNetwork object definition

6.2.1 Creating a layered network topology

A layered queueing network consists of four types of elements: processors, tasks, entries and activities. An entry is a class of service specified through a finite sequence of activities, and hosted by a task running on a (physical) processor. A task is typically a software queue that models access to the capacity of the underpinning processor. Activities model either demands required at the underpinning processor, or calls to entries exposed by some remote tasks.

In the `LayeredNetwork` class, the terms *host* and *processor* are entirely interchangeable.

To create our first layered network, we instantiate a new model as

```
val model = LayeredNetwork("myLayeredModel")
```

We now proceed to instantiate the static topology of processors, tasks and entries:

```
val P1 = Processor(model, "P1", 1, SchedStrategy.PS)
val P2 = Processor(model, "P2", 1, SchedStrategy.PS)
val T1 = Task(model, "T1", 5, SchedStrategy.REF).on(P1)
```

```
val T2 = Task(model, "T2", Int.MAX_VALUE, SchedStrategy.INF).on(P2)
val E1 = Entry(model, "E1").on(T1)
val E2 = Entry(model, "E2").on(T2)
```

An equivalent way to specify the above example is to use the `Host` class instead than the `Processor` class, with identical parameters.

In the above code, the `on` method specifies the associations between the elements, e.g., task `T1` runs on processor `P1`, and accepts calls to entry `E1`. Furthermore, the multiplicity of `T1` is 5, meaning that up to 5 calls can be simultaneously served by this element (i.e., 5 is the multiplicity of servers in the underpinning queueing system for `T1`).

Both processors and tasks can be associated to the standard LINE scheduling strategies. For instance, `T2` will process incoming requests in parallel according as an infinite server node, since we selected the `SchedStrategy.INF` scheduling policy. An exception is that `SchedStrategy.REF` should be used to denote the reference task (e.g. a node representing the clients of the models), which has a similar meaning to the reference node in the object.

6.2.2 Describing host demands of entries

The demands placed by an entry on the underpinning host (also called in layered queueing networks the *host demand*) is described in terms of execution of one or more activities. Although in tools such as LQNS activities can be associated to either entries or tasks, LINE supports only the more general of the two options, i.e., the definition of activities at the level of tasks. In this case:

- Every task defines a collection of activities.
- Every entry needs to specify an initial activity where the execution of the entry starts (the activity is said to be "bound to the entry") and a replying activity, which upon completion terminates the execution of the entry.

For example, in our running example, we may now associate an activity to each entry as follows:

```
val A1 = Activity(model, "A1", Exp(1.0)).on(T1).boundTo(E1).synchCall(E2,3.5)
val A2 = Activity(model, "A2", Exp(2.0)).on(T2).boundTo(E2).repliesTo(E2)
```

Here, `A1` is a task activity for `T1`, acts as initial activity for `E1`, consumes an exponential distributed time on the processor underpinning `T1`, and requires on average 3.5 synchronous calls to `E2` to complete. Each call to entry `E2` is served by the activity `A2`, with a demand on the processor hosting `T2` given by an exponential distribution with rate $\lambda = 2.0$.

Activity graphs

Often, it is useful to structure the sequence of activities carried out by an entry in a graph. Activity graphs can be characterized by precedence relationships of the following kinds:

- *sequence*: two activities are executed sequentially, one after each other. This is implemented through the `ActivityPrecedence.Serial` construct.
- *loop*: an activity is repeated a number of times. This is implemented in `ActivityPrecedence.Loop`.
- *and-fork*: a serial execution is forked into concurrent activities. This can be materialized using the `ActivityPrecedence.AndFork` construct.
- *or-fork*: the server chooses probabilistically which activity to execute next among a set of alternatives. This is implemented in `ActivityPrecedence.OrFork`.
- *and-join*: concurrent activities are joined into a single serial execution. This is implemented in `ActivityPrecedence.AndJoin`.
- *or-join*: merge point for alternative activities that may execute in parallel after a *or-fork*. This is implemented in `ActivityPrecedence.OrJoin`.
- *cache-access*: split point for cache hit/cache miss results in an activity graph. This is implemented in `ActivityPrecedence.CacheAccess`. For usage examples, see `cache_repl_lru` and `cache_compare_repl` in the `examples/` folder.

A composite example showing fork/join precedences and loops is given in `lqn_workflows` in the `examples/` folder.

For instance, we may replace in the running example the specification of the activities underpinning a call to `E2` as

```
val A20 = Activity(model, "A20", Exp(1.0)).on(T2).boundTo(E2)
val A21 = Activity(model, "A21", Erlang.fitMeanAndOrder(1.0,2)).on(T2)
val A22 = Activity(model, "A22", Exp(1.0)).on(T2).repliesTo(E2)

T2.addPrecedence(ActivityPrecedence.Serial(A20, A21, A22))
```

such that a call to `E2` serially executes `A20`, `A21`, and `A22` prior to replying. Here, `A21` is chosen to be an Erlang distribution with given mean (1.0) and number of phases (2).

6.2.3 Debugging and visualization

The structure of a `LayeredNetwork` object can be graphically visualized as follows

```
model.view();
```

```
model.view();
```

The `jsimgView` and `jsimwView` methods can be used to visualize in JMT each layer. This can be done by first calling the `getLayers` method to obtain a list consisting of the Network objects, each one

corresponding to a layer, and then invoking the `jsimgView` and `jsimwView` methods on the desired layer. This is discussed in more details in the next section.

Lastly, we note a number of specification issues that trigger errors in the LQN definition:

Error Type	**Error Type**
Activity in REF task replies	Entry called both synchronously and asynchronously
Entry on task calls itself	Repeated definition of parent task
Entry on task calls entry on the same task	Invalid .on() argument for an activity
Cycle in activity graph	Invalid .on() argument for a task
Unsupported replyTo	Repeated synch calls
Activity with boundTo specification	Repeated asynch calls

6.3 Internals

6.3.1 Representation of the model structure

It is possible to access the internal representation of a `LayeredNetwork` model in a similar way as for objects, i.e.:

```
val lqn: LayeredNetworkStruct = model.getStruct()
```

The return `lqn` structure, of class `LayeredNetworkStruct`, contains all the information about the specified model. It relies on relative and absolute indexing for the elements of the `LayeredNetwork`.

- A *relative* index is a number between 1 and the number of similar elements in the model, e.g., for a model with 3 tasks, the relative index t of a task would be a number in $[1, 3]$.
- An *absolute* index is a number between 1 and the total number of elements (of any kind, except calls) in the model, e.g., for a model with 2 hosts, 3 tasks, 5 entries, and 8 activities, the total number of elements is `nidx`$= 18$ and last activity a may have an absolute index `aidx`$= 18$ and a relative index `a`$= 8$.
- The difference between the relative and the absolute index of an element is referred to as *shift*, e.g., in the previous example `ashift`$= 18 - 8 = 10$.
- Absolute and relative indexing for calls and hosts are identical, call index `cidx` ranges in $[1, ncalls]$ and host index `hidx` ranges in $[1, nhosts]$.

Using the above convention, the internal representation of the model is described in Table 6.1. As in the examples above, relative and absolute indexes are differentiated by using the suffix `idx` in the latter (e.g., `a` vs. `aidx`). This indexing style is used throughout the codebase as well.

The `LayeredNetworkStruct` class in the JAR version provides the internal representation using Java/Kotlin data structures. Table 6.1 lists the main properties available. The `LayeredNetworkStruct` class provides idiomatic Kotlin access to the internal representation with improved type safety and nullable annotations. Table 6.1 lists the main properties available in the Kotlin API.

Table 6.1: `LayeredNetworkStruct` static properties (JAR version)

Field	Type	Description
`nidx`	`int`	Total number of `LayeredNetwork` elements
`nhosts`	`int`	Number of `Hosts` or `Processor` elements
`ntasks`	`int`	Number of `Tasks` elements
`nentries`	`int`	Number of `Entry` elements
`nacts`	`int`	Number of `Activity` elements
`ncalls`	`int`	Number of calls issued by `Activity` elements
`hshift`	`int`	For host h, the value h+`hshift` returns its absolute index in 1...`nidx`
`tshift`	`int`	For task t, the value t+`tshift` returns its absolute index in 1...`nidx`
`eshift`	`int`	For entry e, the value e+`eshift` returns its absolute index in 1...`nidx`
`ashift`	`int`	For activity a, the value a+`ashift` returns its absolute index in 1...`nidx`
`cshift`	`int`	For call c, the value c+`cshift` returns its absolute index in 1...`ncalls`
`tasksof`	`Map⟨Integer, List⟨Integer⟩⟩`	Map from host absolute index to list of task absolute indexes on that host
`entriesof`	`Map⟨Integer, List⟨Integer⟩⟩`	Map from task absolute index to list of entry absolute indexes on that task
`actsof`	`Map⟨Integer, List⟨Integer⟩⟩`	Map from entry/task absolute index to list of activity absolute indexes
`callsof`	`Map⟨Integer, List⟨Integer⟩⟩`	Map from activity absolute index to list of call absolute indexes
`hostdem`	`Map⟨Integer, Distribution⟩`	Host demand distribution for each element by absolute index
`think`	`Map⟨Integer, Distribution⟩`	Think time distribution for each element by absolute index
`sched`	`Map⟨Integer, SchedStrategy⟩`	Scheduling strategy for host or task by absolute index
`names`	`Map⟨Integer, String⟩`	Name of element by absolute index
`hashnames`	`Map⟨Integer, String⟩`	Name with type prefix by absolute index ("H:" host, "R:" reference task, "T:" task, "C:" cache task, "E:" entry, "I:" item entry, "A:" activity)
`mult`	`Matrix`	Multiplicity for host or task by absolute index
`maxmult`	`Matrix`	Maximum multiplicity for host or task by absolute index
`repl`	`Matrix`	Replication factor for host or task by absolute index
`type`	`Matrix`	`LayeredNetworkElement` type id for element by absolute index
`nitems`	`Matrix`	Number of items in `CacheTask` or `ItemEntry` by absolute index
`itemcap`	`Map⟨Integer, Integer⟩`	Cache capacity for cache list by absolute index
`replacestrat`	`Matrix`	`ReplacementStrategy` id for cache task by absolute index

Continued on next page

Table 6.1 – *Continued from previous page*

Field	Type	Description
itemproc	Map⟨Integer, DiscreteDistribution⟩	Item popularity distribution for ItemEntry by absolute index
calltype	Map⟨Integer, CallType⟩	CallType for call by call index
callpair	Matrix	Call relationship matrix: column 1 = activity issuing call, column 2 = entry being called
callproc	Map⟨Integer, DiscreteDistribution⟩	Number of calls distribution by call absolute index
callnames	Map⟨Integer, String⟩	Name of call by call absolute index
callhashnames	Map⟨Integer, String⟩	Call name with type prefixes for source and destination
actpretype	Matrix	ActivityPrecedenceType id before activity by absolute index
actposttype	Matrix	ActivityPrecedenceType id after activity by absolute index
graph	Matrix	Adjacency matrix: $\neq 0$ if element i "runs on", "calls" or "precedes" element j
dag	Matrix	Directed acyclic graph version of graph with flipped entry-task edges
parent	Matrix	Parent element absolute index for each element
replygraph	Matrix	True if activity replies, ending the entry call
taskgraph	Matrix	True if host/task i calls host/task j
iscache	Matrix	True if task is a CacheTask
iscaller	Matrix	True if element i calls element j
issynccaller	Matrix	True if element i issues synchronous call to element j
isasynccaller	Matrix	True if element i issues asynchronous call to element j
isref	Matrix	True if task is a reference task
isfunction	Matrix	True if task is a function task
setuptime	Map⟨Integer, Distribution⟩	Setup time distribution by absolute index
delayofftime	Map⟨Integer, Distribution⟩	Delay-off time distribution by absolute index
conntasks	Matrix	Connected tasks matrix
hitmissaidx	List⟨Integer⟩	List of hit/miss activity absolute indexes
hitaidx	Integer	Hit activity absolute index
missaidx	Integer	Miss activity absolute index

6.3.2 Decomposition into layers

Layers are a form of decomposition where we model the performance of one or more servers. The activity of clients not detailed in that layer is taken into account through an artificial delay station, placed in a closed loop to the servers [42]. This artificial delay is used to model the inter-arrival time between calls issued by that client.

The current version of LINE adopts SRVN-type layering [24], whereby a layer corresponds to one and only one resource, either a processor or a task. The getLayers method returns a cell array consisting of the objects corresponding to each layer

```
\texttt{List\textlangle Network\textrangle} layers = model.getLayers()
```

The decomposition is performed through the LN solver described later.

Within each layer, classes are used to model the time a job spends in a given activity or call, with synchronous calls being modeled by classed with label including an arrow, e.g., is a closed class used represent synchronous calls from activity AS1 to entry E3, whereas denotes an asynchronous call. Artificial delays and reference nodes are modelled as a delay station named 'Clients', whereas the task or processor assigned to the layer is modelled as the other node in the layer.

6.4 Solvers

LINE offers two solvers for the solution of a LayeredNetwork model consisting in its own native solver (LN) and a wrapper (LQNS) to the LQNS solver [24]. The latter requires a distribution of LQNS to be available on the operating system command line.

The solution methods available for LayeredNetwork models are similar to those for Network objects. For example, the avgTable can be used to obtain a full set of mean performance indexes for the model, e.g.,

```
val avgTable = SolverLQNS(model).avgTable()
avgTable.print()
```

Note that in the above table, some performance indexes are marked as NaN because they are not defined in a layered queueing network. Further, compared to the avgTable method in objects, LayeredNetwork do not have an explicit differentiation between stations and classes, since in a layer a task may either act as a server station or a client class.

The main challenge in solving layered queueing networks through analytical methods is that the parameterization of the artificial delays depends on the steady-state performance of the other layers, thus causing a cyclic dependence between input parameters and solutions across the layers. Depending on the solver in use, such issue can be addressed in a different way, but in general a decomposition into layers will remain parametric on a set of response times, throughputs and utilizations.

This issue can be resolved through solvers that, starting from an initial guess, cyclically analyze the layers and update their artificial delays on the basis of the results of these analyses. Both LN and LQNS implement this solution method. Normally, after a number of iterations the model converges to a steady-state solution, where the parameterization of the artificial delays does not change after additional iterations.

6.4.1 LQNS

The LQNS wrapper operates by first transforming the specification into a valid LQNS XML file. Subsequently, LQNS calls the solver and parses the results from disks in order to present them to the user in the appropriate LINE tables or vectors. The options.method can be used to configure the LQNS execution as follows:

- options.method=std or lqns: LQNS analytical solver with default settings.

- `options.method=exact`: the solver will execute the standard LQNS analytical solver with the exact MVA method.
- `options.method=srvn`: LQNS analytical solver with SRVN layering.
- `options.method=srvnexact`: the solver will execute the standard LQNS analytical solver with SRVN layering and the exact MVA method.
- `options.method=lqsim`: LQSIM simulator, with simulation length specified via the `samples` field (i.e., with parameter `-A options.samples, 0.95`).

Upon invocation, the `lqns` or `lqsim` commands will be searched for in the system path. If they are unavailable, the termination of `SolverLQNS` will interrupt.

6.4.2 QNS

LINE also includes a dedicated wrapper solver for the `qnsolver` utility distributed within LQNS, called `SolverQNS`. This allows users to evaluate product-form models using the MVA algorithms implemented within LQNS. The available options specify the multiserver handling algorithm:

- `options.method=conway`: Conway's multiserver approximation within the Linearizer algorithm proposed in [17].
- `options.method=rolia`: Rolia's multiserver in the Methods of Layers paper [42].
- `options.method=reiser`: load-dependent mean-value analysis as described originally by Reiser-Lavenberg in [40].
- `options.method=zhou`: Zhou-Woodside's multiserver approximation in [50].

6.4.3 LN

The native `LN` solver iteratively applies the layer updates until convergence of the steady-state measures. Since updates are parametric on the solution of each layer, `LN` can apply any of the solvers described in the solvers chapter to the analysis of individual layers, as illustrated in the following example for the `MVA` solver

```
val options = SolverLN.defaultOptions()
val mvaopt = SolverMVA.defaultOptions()
SolverLN(model, { layer -> SolverMVA(layer, mvaopt) }, options).avgTable().print()
```

Options parameters may also be omitted. The `LN` method converges when the maximum relative change of mean response times across layers from the last iteration is less than `options.iter_tol`.

Methods supported by the LN solver include:

- `options.method=default`: default recursive solution based on mean values
- `options.method=moment3`: solution by recursive 3-moment approximation of response time distributions.

6.5 Model import and export

A `LayeredNetwork` can be easily read from, or written to, a XML file based on the LQNS meta-model format[1]. The read operation can be done using a static method of the `LayeredNetwork` class, i.e.,

```
val model: LayeredNetwork = LayeredNetwork.parseXML(filename)
```

Conversely, the write operation is invoked directly on the model object

```
model.writeXML(filename)
```

In both examples, `filename` is a string including both file name and its path.

Finally, we point out that it is possible to export a LQN in the legacy SRVN file format[2] by means of the `writeSRVN(filename)` function.

[1] https://raw.githubusercontent.com/layeredqueuing/V5/master/xml/lqn.xsd
[2] http://www.sce.carleton.ca/rads/lqns/lqn-documentation/format.pdf

Chapter 7

Random environments

Systems modeled with LINE can be described as operating in an environment with a state that affects the way the system dynamics. To distinguish the states of the environment from the ones of the system within it, we shall refer to the former as the environment *stages*. In particular, LINE 3.0.x supports the definition of a class of random environments subject to three assumptions:

- The stage of the environment evolves independently of the state of the system.
- The dynamics of the environment stage can be described by a continuous-time Markov chain.
- The topology of the system is independent of the environment stage.

The above definitions are in particular appropriate to describe systems specified by input parameters (e.g., service rates, scheduling weights, etc) that change with the environment stage. For example, an environment with two stages, say normal load and peak load, may differ for the number of servers that are available in a queueing station, i.e., the system controller may add more servers during peak load. Upon a stage change in the environment, the model parameters will instantaneously change, and the system state reached during the previous stage will be used to initialize the system in the new stage.

Although in a number of cases the system performance may be similar to a weighted combination of the average performance in each stage, this is not true in general, especially if the system dynamic (i.e., the rate at which jobs arrive and get served) and the environment dynamic (i.e., the rate at which the environment changes active stage) have a similar magnitude [14].

7.1 Environment object definition

7.1.1 Specifying the environment

In LINE, an environment is internally described by a Markov renewal process (MRP) with transition times belonging to the `Markovian` class. A MRP is similar to a Markov chain, but state transitions are not

restricted to be exponential. Although the time spent in each state of the MRP is not exponential, the MRP with phase-type transitions can be easily transformed into an equivalent continuous-time Markov chain (CTMC) to enable analysis, a task that LINE performs automatically.

To specify an environment, we first create an `Env` object with the environment name

```
val E = 2
val envModel = Env("UnreliableEnv", E)
```

where the `E` parameter indicates the number of stages in the environment. We then add two stages

```
envModel.addStage("Online", "UP", network1)
envModel.addStage("Offline", "DOWN", network2)
```

where the constructor specifies the stage name, an arbitrary string to classify the stage (here taken from a taxonomy in the `Semantics` class), follows by a object describing the system model conditional on the environment being in the corresponding stage.

We now describe that the transitions between stages are both exponential, with different rates

```
envModel.addTransition(0, 1, Exp(1))
envModel.addTransition(1, 0, Exp(2))
```

We can also add a self-loop on the online stage as follows

```
envModel.addTransition(0, 0, Erlang.fitMeanAndOrder(1,2))
```

which would cause a race condition between two distributions in stage two: the exponential transition back to the offline stage, and the Erlang-2 distributed transition with unit rate that remains in the online stage. The underpinning CTMC will therefore consider the distribution of the minimum between the exponential and the Erlang-2 distribution, in order to decide the next stage transition. State space explosion may occur in the definition of an environment if the user specifies a large number of non-exponential transition. For example, a race condition among n Erlang-2 distribution translates at the level of the CTMC into a state space with 2^n states. In such situations, it is recommended to replace some of the distributions with exponential ones.

To summarize the properties of the environment defined above we may use the `getStageTable` method

```
// Print stage table information
envModel.printStageTable()
// Output is identical to Java version above
```

In the table, the `State` column gives a numerical identifier for each stage, followed by its stage probability at equilibrium, a Markovian representation of the time spent in it before a transition, and by a pointer to the sub-model associated to that stage.

7.1.2 Specifying a reset policy

When the environment transitions, the default policy is that the associated model is re-initialized using the marginal queue-length values observed at departure instants. This means in practice that jobs in execution at a server are required all to restart execution at that server upon occurrence of a transition. This may not be possible in some models, for example when a station is removed from the model. In that case, one can define a custom reset policy by instantiating transitions as, e.g.,

```
// Define a reset function that moves all jobs into station 0
val resetRule = Env.ResetQueueLengthsFunction { qExit ->
    val numStations = qExit.numRows
    val numClasses = qExit.numCols
    val qReset = Matrix(numStations, numClasses)

    // Move all jobs to station 0, preserving their classes
    for (c in 0 until numClasses) {
        var totalJobsInClass = 0.0
        for (s in 0 until numStations) {
            totalJobsInClass += qExit[s, c]
        }
        qReset[0, c] = totalJobsInClass
    }
    qReset
}

// Add transition with reset rule
// Assuming stages are indexed (0 for 'Online', 1 for 'Offline')
envModel.add_transition(0, 1, Exp(1.0), resetRule)
```

In Kotlin, the `resetRule` is defined as a lambda expression implementing the `ResetQueueLengthsFunction` interface. The reset policy works identically to the Java version, moving all jobs to station 0 while preserving their class assignments.

7.1.3 Specifying system models for each stage

LINE places loose assumptions in the way the system should be described in each stage. It is just expected that the user supplies a model object, either a or a `LayeredNetwork`, in each stage, and that a transient analysis method is available in the chosen solver, a requirement fulfilled for example by `SolverFluid`.

However, we note that the model definition can be somewhat simplified if the user describes the system model in a separate MATLAB function, accepting the stage-specific parameters in input to the function. This enables reuse of the system topology across stages, while creating independent model objects.

7.2 Solvers

The steady-state analysis of a system in a random environment is carried out in LINE using the blending method [14], which is an iterative algorithm leveraging the transient solution of the model. In essence, the model looks at the *average* state of the system at the instant of each stage transition, and upon restarting the system in the new stage re-initializes it from this average value. This algorithm is implemented in LINE by the `SolverEnv` class, which is described next.

7.2.1 ENV

The `SolverEnv` class applies the blending algorithm by iteratively carrying out a transient analysis of each system model in each environment stage, and probabilistically weighting the solution to extract the steady-state behavior of the system.

As in the transient analysis of objects, LINE does not supply a method to obtain mean response times, since Little's law does not hold in the transient regime. To obtain the mean queue-length, utilization and throughput of the system one can call as usual the `avg` method on the `SolverEnv` object, e.g.,

```
val solvers = arrayOfNulls<NetworkSolver>(E)
for (e in 0 until E) {
  solvers[e] = SolverFluid(envModel.getModel(e))
  solvers[e]!!.options = SolverOptions(SolverType.Fluid)
}

val envSolver = SolverEnv(envModel, solvers, options)
envSolver.avg()
val result = envSolver.result
```

Note that as model complexity grows, the number of iterations required by the blending algorithm to converge may grow large. In such cases, the `options.iter_max` option may be used to bound the maximum analysis time.

Bibliography

[1] David F Anderson. A modified next reaction method for simulating chemical systems with time dependent propensities and delays. *The Journal of chemical physics*, 127(21), 2007.

[2] S. Balsamo. Product form queueing networks. In Günter Haring, Christoph Lindemann, and Martin Reiser, editors, *Performance Evaluation: Origins and Directions*, volume 1769 of *Lecture Notes in Computer Science*, pages 377–401. Springer, 2000.

[3] M. Bertoli, G. Casale, and G. Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *Proc. of the 40th Annual Simulation Symposium (ANSS)*, pages 3–10, 2007.

[4] 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.

[5] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[6] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6:219–234, 1986.

[7] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service stations. *Performance Evaluation*, 4:241–260, 1984.

[8] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Trans. Software Engineering*, 35(2):162–177, 2009.

[9] G. Casale. Accelerating performance inference over closed systems by asymptotic methods. In *Proc. of ACM SIGMETRICS*. ACM Press, 2017.

[10] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with Line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, dec 2020.

[11] G. Casale, P.G. Harrison, and O.W. Hong. Facilitating load-dependent queueing analysis through factorization. *Perform. Eval.*, 2021.

[12] G. Casale, Richard R. Muntz, and Giuseppe Serazzi. Geometric bounds: A noniterative analysis technique for closed queueing networks. *IEEE Trans. Computers*, 57(6):780–794, 2008.

[13] G. Casale, J. F. Pérez, and W. Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. In *Proceedings of IFIP PERFORMANCE*, 2015.

[14] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

[15] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queuing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[16] W.-M. Chow. Approximations for large scale closed queueing networks. *Perform. Eval*, 3(1):1–12, 1983.

[17] A. E. Conway. *Fast Approximate Solution of Queueing Networks with Multi-Server Chain-Dependent FCFS Queues*, pages 385–396. Springer US, Boston, MA, 1989.

[18] A. E. Conway and N. D. Georganas. RECAL - A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33(4):768–791, 1986.

[19] E. de Souza e Silva and R. R. Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Trans. Computers*, 39(6):840–842, 1990.

[20] R.-A. Dobre, Z. Niu, and G. Casale. Approximating fork-join systems via mixed model transformations. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 273–280, New York, NY, USA, 2024. Association for Computing Machinery.

[21] D. L. Eager and J. N. Lipscomb. The AMVA priority approximation. *Perform. Eval.*, 8(3):173–193, 1988.

[22] G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton, 1996.

[23] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Engineering*, 35(2):148–161, 2009.

[24] G. Franks, P. Maly, C. M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. *Layered Queueing Network Solver and Simulator User Manual*, 2012.

[25] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *Queueing Syst*, 83(3-4):293–328, 2016.

[26] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[27] A. Harel, S. Namn, and J. Sturm. Simple bounds for closed queueing networks. *Queueing Systems*, 31(1-2):125–135, 1999.

[28] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Computers*, 100(11):1099–1109, 1982.

[29] G. Horváth and M. Telek. Butools 2: A rich toolbox for markovian performance evaluation. In *Proc. of VALUETOOLS*, pages 137–142, ICST, Brussels, Belgium, Belgium, 2017. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[30] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41(4):480–488, 1992.

[31] S. S. Lavenberg. A perspective on queueing models of computer performance. *Perform. Eval.*, 10(1):53–76, 1989.

[32] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[33] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks (work in progress paper). In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 34–39, New York, NY, USA, 2024. Association for Computing Machinery.

[34] 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.

[35] J. McKenna and D. Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed markovian networks. *J. ACM*, 31(2):346–360, April 1984.

[36] J. F. Pérez and G. Casale. Assessing SLA compliance from Palladio component models. In *Proceedings of the 2nd MICAS*, 2013.

[37] J. F. Pérez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Trans. Reliability*, 66(3):837–853, Sept 2017.

[38] M. Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Trans. Communications*, 27(8):1199–1209, 1979.

[39] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perform. Eval.*, 1:7–18, 1981.

[40] M. Reiser and S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27:313–322, 1980.

[41] T. G. Robertazzi. *Computer Networks and Systems*. Springer, 2000.

[42] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Engineering*, 21(8):689–700, August 1995.

[43] J. Ruuskanen, T. Berner, K.-E. Årzén, and A. Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Perform. Evaluation*, 151:102231, 2021.

[44] P. J. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proc. of the Int'l Conf. on Stoch. Control and Optim.*, pages 25–29, Amsterdam, 1979.

[45] A. Seidmann, P. J. Schweitzer, and S. Shalev-Oren. Computerized closed queueing network models of flexible manufacturing systems: A comparative evaluation. *Large Scale Systems*, 12:91–107, 1987.

[46] K. Sevcik. Priority scheduling disciplines in queuing network models of computer systems. In *IFIP Congress*, 1977.

[47] W. Wang, G. Casale, and C. A. Sutton. A bayesian approach to parameter inference in queueing networks. *ACM Trans. Model. Comput. Simul.*, 27(1):2:1–2:26, 2016.

[48] M. Woodside. *Tutorial Introduction to Layered Modeling of Software Performance*. Carleton University, February 2013.

[49] J. Zahorjan, D. L. Eager, and H. M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Perform. Eval.*, 8(4):255–270, 1988.

[50] S. Zhou and M. Woodside. A multiserver approximation for cloud scaling analysis. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ICPE '22, page 129–136, New York, NY, USA, 2022. Association for Computing Machinery.

Appendix A

Examples

Table A.1: Examples

Example	Problem
cache_replc_rr	A small cache model with an open arrival process
cache_replc_fifo	A small cache model with a closed job population
cache_replc_lru	A layered network with a caching layer
cache_compare_replc	A layered network with a caching layer having a multi-level cache
cache_replc_routing	A caching model with state-dependent output routing
cdf_respt_closed	Station response time distribution in a single-class single-job closed network
cdf_respt_closed_threeclasses	Station response time distribution in a multi-chain closed network
cdf_respt_open_twoclasses	Station response time distribution in a multi-chain open network
cdf_respt_distrib	Simulation-based station response time distribution analysis
cdf_respt_populations	Station response time distribution under increasing job populations
cqn_repairmen	Solving a single-class exponential closed queueing network
cqn_twoclass_hyperl	Solving a closed queueing network with a multi-class FCFS station
cqn_threeclass_hyperl	Solving exactly a multi-chain product-form closed queueing network
cqn_multiserver	Local state space generation for a station in a closed network
cqn_oneline	1-line exact MVA solution of a cyclic network of PS and INF stations
cqn_twoclass_erl	Closed network with round robin scheduling
cqn_bcmp_theorem	Comparison of different scheduling policies that preserve the product-form solution
cqn_repairmen_multi	Multi-server closed queueing network with repairmen
cqn_twoqueues_multi	Closed queueing network with two multi-server queues
cqn_twoqueues	Simple closed network with two queues
cs_implicit	Class switching with implicit routing
cs_multi_diamond	Class switching with multiple diamond patterns
cs_single_diamond	Class switching with single diamond pattern
cs_transient_class	Class switching with transient classes
fj_basic_open	A simple single class open fork-join network
fj_twoclasses_forked	A multiclass open fork-join network
fj_basic_nesting	A closed model with nested forks and joins

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
fj_nojoin	An open model with a fork but without a join
fj_basic_closed	A simple single class closed fork-join network
fj_serialfjs_open	Two open fork-joins subsystems in tandem
fj_cs_postfork	Two-class fork-join with a class that switches into the other after the fork
fj_cs_multi_visits	Two fork-joins loops within the same chain
fj_route_overlap	A model with overlapping routes in a fork-join network
fj_asymm	Asymmetric fork-join network
fj_delays	Fork-join network with delays
fj_complex_serial	Complex serial fork-join network
fj_threebranches	Fork-join network with three branches
fj_cs_prefork	Fork-join network with class-switching before fork
fj_deep_nesting	Fork-join network with deep nesting
fj_serialfjs_closed	Closed serial fork-join network
init_state_fcfs_exp	Specifying an initial state and prior in a single class model.
init_state_fcfs_nonexp	Specifying an initial state and prior in a multiclass model.
init_state_ps	Specifying an initial state and prior in a model with class-switching.
lqn_serial	Analyze a layered network specified in a LQNS XML file
lqn_multi_solvers	Specifying and solving a basic layered network
lqn_init	Specifying and solving a basic layered network with initialization
lqn_twotasks	Layered network with two tasks
lqn_bpmn	BPMN to layered network transformation
lqn_workflows	Workflow modeling in layered networks
lqn_function	Layered network function modeling
lqn_basic	Basic layered network example
ld_multiserver_fcfs	Solving a single-class load-depedent closed model
ld_multiserver_ps_twoclasses	Solving a two-node multiclass load-depedent closed model
ld_multiserver_ps	Solving a three-node multiclass load-depedent closed model
ld_class_dependence	Load-dependent model with class dependence
cqn_scheduling_dps	Parameterization of a discriminatory processor sharing (DPS) station
cqn_mmpp2_service	Automatic detection of solvers that cannot analyze the model
mqn_basic	Solving a queueing network model with both closed and open classes
mqn_multiserver_ps	A difficult mixed model with sparse routing among multi-server nodes
mqn_multiserver_fcfs	Mixed model with multiserver FCFS nodes
mqn_singleserver_fcfs	Mixed model with single server FCFS
mqn_singleserver_ps	Mixed model with single server PS
oqn_basic	Solving a queueing network model with open classes, scalar cutoff options
oqn_oneline	1-line solution of a tandem network of PS and INF stations
oqn_cs_routing	Solving a queueing network model with open classes, matrix cutoff options
oqn_trace_driven	Trace-driven simulation of an M/M/1 queue
oqn_vsinks	A model illustrating the emulation of multiple sinks
oqn_fourqueues	A large multiclass example with PS and FCFS
prio_hol_open	A multiclass example with PS, SIRO, FCFS, HOL priority
prio_hol_closed	A high-load multiclass example with PS, SIRO, FCFS, HOL priority
prio_psprio	A repaimen model with PS priority scheduling.
prio_identical	Priority model with identical classes
renv_twostages_repairmen	Solving a model in a 2-stage random environment with exponential rates

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
renv_fourstages_repairmen	Solving a model in a 4-stage random environment with Coxian rates
renv_threestages_repairmen	Solving a model in a 3-stage random environment with Erlang rates
sdroute_closed	A model with round-robin routing
sdroute_twoclasses_closed	A model with round-robin routing after multi-class PH and MAP service
sdroute_open	A load-balancer modeled as a router
statepr_aggr	Computing marginal state probabilities for a node
statepr_aggr_large	Computing marginal state probabilities for a node under class-switching
statepr_sys_aggr	Computing joint state probabilities for a system with two nodes under class-switching
statepr_sys_aggr_large	Computing joint state probabilities under class-switching and with delay nodes
statepr_allprobs_ps	Computing probabilities under PS class-switching and with delay nodes
statepr_allprobs_fcfs	Computing probabilities under PS and FCFS class-switching and with delay nodes
spn_basic_open	JMT simulation of a simple stochastic Petri net model
spn_open_sevenplaces	JMT simulation of a complex stochastic Petri net model
spn_twomodes	Stochastic Petri net with two modes
spn_fourmodes	Stochastic Petri net with four modes
spn_inhibiting	Stochastic Petri net with inhibiting transitions
spn_closed_fourplaces	Closed Stochastic Petri net with four places
spn_closed_twoplaces	Closed Stochastic Petri net with two places
spn_basic_closed	Basic closed stochastic Petri net
tut_01_mm1_basics	M/M/1 queue basics
tut_02_mg1_multiclass_solvers	M/G/1 multiclass solvers
tut03_repairmen	Repairmen model
tut_04_lb_routing	Load balancing routing
tut05_completes_flag	Complete flag usage
tut_06_cache_lru_zipf	Cache LRU Zipf distribution
tut_07_respt_cdf	Response time CDF analysis
tut_08_opt_load_balancing	Optimal load balancing
tut_09_dep_process_analysis	Dependent process analysis
lcq_singlehost	Single host layered cache queueing
lcq_threehosts	Three hosts layered cache queueing
swt_basic	Basic switchover times model
polling_exhaustive_exp	Exhaustive polling with exponential times
polling_gated	Gated polling
polling_klimited	K-limited polling
polling_exhaustive_det	Exhaustive polling with deterministic times

Appendix B

API Function Reference

This appendix provides a comprehensive catalog of all API functions available in the `jline.api` package. The table lists each function name, its organizational package, and a brief description of its purpose.

Table B.1: Complete API Function Reference

Function Name	Package	Description
`amap2_fit_gamma`	mam	Fits AMAP(2) distributions to match moments and correlation characteristics
`amap2_fit_gamma_map`	mam	Fits AMAP(2) by approximating arbitrary-order MAP with preserved correlation structure
`amap2_fit_gamma_trace`	mam	Fits AMAP(2) from empirical traces while preserving autocorrelation characteristics
`aph2_adjust`	mam	Adjusts moments to ensure feasibility bounds for APH(2) fitting procedures
`aph2_assemble`	mam	Constructs APH(2) transition matrices from specified rates and transition probabilities
`aph2_fit`	mam	Fits APH(2) distributions to match given moments with automatic feasibility adjustment
`aph2_fit_map`	mam	Fits APH(2) distributions by approximating arbitrary-order MAP processes
`aph2_fit_trace`	mam	Fits APH(2) distributions from empirical inter-arrival time traces
`aph2_fitall`	mam	Fits multiple APH(2) distributions to match given moments with exhaustive parameter search
`aph_bernstein`	mam	Constructs APH distributions using Bernstein exponential approximation methods
`aph_fit`	mam	Fits APH distributions to specified moments using optimization and approximation techniques
`aph_rand`	mam	APH random generation algorithms
`aph_simplify`	mam	Simplifies and combines APH distributions using structural pattern operations

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
cache_erec	cache	Implements exact recursive (EREC) algorithms for cache system analysis
cache_gamma	cache	Computes cache access factors from request arrival rates and routing matrices
cache_gamma_lp	cache	Computes cache access factors using linear programming optimization methods
cache_miss	cache	Provides general-purpose algorithms for computing cache miss rates across
cache_miss_asy	cache	Provides asymptotic approximation methods for cache miss rate analysis
cache_miss_fpi	cache	Computes cache miss probabilities using fixed-point iteration methods
cache_miss_rayint	cache	Estimates cache miss rates using ray method for partial differential equations
cache_mva	cache	Implements Mean Value Analysis algorithms for cache system performance
cache_mva_miss	cache	Implements Mean Value Analysis (MVA) algorithms for computing cache miss
cache_prob_erec	cache	Computes exact cache state probabilities using recursive methods based on
cache_prob_fpi	cache	Computes cache state probabilities using fixed-point iteration algorithms
cache_prob_rayint	cache	Computes cache state probabilities using ray integration methods for
cache_rayint	cache	Implements ray integration techniques for cache system analysis using
cache_rrm_meanfield_ode	cache	Implements the mean field ordinary differential equation system for Random
cache_t_hlru	cache	Computes response time metrics for Hierarchical Least Recently Used (H-LRU)
cache_t_lrum	cache	Computes response time metrics for Least Recently Used with Multiple servers
cache_t_lrum_map	cache	Analyzes response times for LRUM cache systems with Markovian Arrival
cache_ttl_hlru	cache	Implements TTL approximation for Hierarchical LRU (H-LRU) cache systems
cache_ttl_lrua	cache	Implementation of Time-To-Live (TTL) approximation for LRU(A) cache systems
cache_ttl_lrum	cache	Implementation of Time-To-Live approximation for Least Recently Used with
cache_ttl_lrum_map	cache	Combines TTL approximation with LRUM cache policies and Markovian Arrival
cache_ttl_tree	cache	Tree-based TTL cache analysis implementation for the LINE solver framework
cache_xi_bvh	cache	Computes cache xi terms using the iterative method from Gast-van Houdt

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
cache_xi_fp	cache	Estimates cache xi terms using fixed-point algorithms. Xi terms represent
ctmc_courtois	mc	Courtois decomposition for nearly completely decomposable CTMCs
ctmc_kms	mc	Koury-McAllister-Stewart aggregation-disaggregation method for CTMCs
ctmc_makeinfgen	mc	Constructs and validates infinitesimal generator matrices for continuous-time
ctmc_multi	mc	Multi-level aggregation method for CTMCs
ctmc_pseudostochcomp	mc	API function for mc operations
ctmc_rand	mc	Generates random infinitesimal generator matrices for continuous-time Markov chains
ctmc_randomization	mc	Converts a continuous-time Markov chain into an equivalent discrete-time chain
ctmc_relsolve	mc	Equilibrium distribution of a continuous-time Markov chain re-normalized with respect to
ctmc_simulate	mc	Generates sample paths for CTMCs using the standard simulation algorithm with
ctmc_solve	mc	Computes the steady-state probability distribution for CTMCs by solving the linear
ctmc_solve_reducible	mc	Solve reducible CTMCs by converting to DTMC via randomization
ctmc_ssg	mc	API function for mc operations
ctmc_ssg_reachability	mc	CTMC State Space Generator for Reachability Analysis
ctmc_stmonotone	mc	Computes the stochastically monotone upper bound for a CTMC
ctmc_stochcomp	mc	Implements stochastic complementarity analysis for CTMCs to identify strongly
ctmc_takahashi	mc	Takahashi's aggregation-disaggregation method for CTMCs
ctmc_testpf_kolmogorov	mc	Test if a CTMC has product form using Kolmogorov's criteria
ctmc_timereverse	mc	Computes the infinitesimal generator of the time-reversed continuous-time
ctmc_transient	mc	Computes transient probabilities for CTMCs using numerical integration of the
ctmc_uniformization	mc	CTMC Transient Analysis via Uniformization
dtmc_isfeasible	mc	Check if a matrix represents a feasible DTMC transition matrix
dtmc_makestochastic	mc	Converts non-negative matrices into valid discrete-time Markov chain transition
dtmc_rand	mc	Generates random stochastic transition matrices for discrete-time Markov chains
dtmc_simulate	mc	Generates sample trajectories for DTMCs by sampling from the transition probability
dtmc_solve	mc	Computes the steady-state probability distribution for DTMCs by converting the
dtmc_solve_reducible	mc	Result class for DTMC solve reducible

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
dtmc_stochcomp	mc	Returns the stochastic complement of a DTMC
dtmc_timereverse	mc	Compute the infinitesimal generator of the time-reversed DTMC
dtmc_uniformization	mc	Result class for DTMC uniformization analysis containing the probability vector and maximum iterations used
lossn_erlangfp	lossn	Implements fixed-point algorithms for analyzing loss networks using Erlang
lsn_max_multiplicity	lsn	Computes maximum multiplicity constraints for load sharing network (LSN)
m3pp22_fitc_approx_cov	mam.m3pp	Implements parameter fitting for second-order Marked Markov Modulated Poisson Process
m3pp22_fitc_approx_cov_multiclass	mam.m3pp	Implements constrained optimization for fitting M3PP(2,2) parameters given an underlying
m3pp22_interleave_fitc	mam.m3pp	Implements lumped superposition of multiple M3PP(2,2) processes using interleaved
m3pp2m_fitc	mam.m3pp	Implements exact fitting of second-order Marked Markov Modulated Poisson Process
m3pp2m_fitc_approx	mam.m3pp	Implements approximation-based fitting for M3PP(2,m) using optimization methods
m3pp2m_fitc_approx_ag	mam.m3pp	Implements auto-gamma approximation method for M3PP(2,m) parameter fitting
m3pp2m_fitc_approx_ag_multiclass	mam.m3pp	Implements multiclass auto-gamma fitting for M3PP(2,m) with variance and covariance
m3pp2m_interleave	mam.m3pp	Implements interleaved superposition of multiple M3PP(2,m) processes to construct
m3pp_interleave_fitc	mam.m3pp	Implements fitting and interleaving of k second-order M3PP processes with varying
m3pp_interleave_fitc_theoretical	mam.m3pp	Implements theoretical MMAP fitting through M3PP interleaving using analytical
m3pp_interleave_fitc_trace	mam.m3pp	Implements M3PP interleaving and fitting directly from empirical trace data
m3pp_rand	mam.m3pp	Implements random generation of Markovian Multi-class Point Processes (M3PP)
m3pp_superpos_fitc	mam.m3pp	Implements superposition-based fitting of k second-order M3PP processes into
m3pp_superpos_fitc_theoretical	mam.m3pp	Implements superposition fitting of k second-order M3PP processes using theoretical
mamap22_fit_gamma_fs_trace	mam	Fits MAMAP(2,2) from trace data using gamma autocorrelation and forward-sigma characteristics
mamap22_fit_multiclass	mam	Fits MAMAP(2,2) processes for two-class systems with forward moments and sigma characteristics
mamap2m_coefficients	mam	Computes coefficients for MAMAP(2,m) fitting formulas in canonical forms
mamap2m_fit	mam	Fits MAMAP(2,m) processes matching moments, autocorrelation, and class characteristics

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mamap2m_fit_fb_multiclass	mam	Fits MAMAP using combined forward and backward moment characteristics for multiclass systems
mamap2m_fit_gamma_fb_mmap	mam	Fits MAMAP with autocorrelation control using forward-backward moments from MMAP input
mamap2m_fit_mmap	mam	Fits MAPH/MAMAP(2,m) by approximating characteristics of input MMAP processes
mamap2m_fit_trace	mam	Fits MAMAP(2,m) processes from empirical trace data with inter-arrival times and class labels
map2_fit	mam	Fits MAP(2) processes to match specified moments and auto-correlation decay rates
map2mmpp	mam	Converts MAP representations to MMPP format for compatibility with MMPP-specific algorithms
map_acf	mam	Computes autocorrelation function (ACF) values for MAP inter-arrival times at specified lags
map_acfc	mam	Computes ACFC values for MAP counting processes over time intervals, measuring temporal correlation
map_block	mam	Constructs MAP(2) representations from moment and auto-correlation parameters using fallback
map_ccdf_derivative	mam	Computes derivatives of MAP complementary cumulative distribution functions at zero
map_cdf	mam	Computes CDF values for MAP inter-arrival times using CTMC uniformization techniques
map_checkfeasible	mam	Comprehensive validation of MAP matrices including stochastic properties, numerical stability,
map_count_mean	mam	Computes mean number of arrivals in MAP counting processes over specified time intervals
map_count_moment	mam	Computes power moments of MAP counting processes using moment generating functions and
map_count_var	mam	Computes variance of MAP counting processes over specified time intervals using matrix
map_embedded	mam	Computes embedded DTMC matrices from MAP representations by extracting transition
map_erlang	mam	Constructs MAP representations of Erlang-k processes with specified means and phases
map_exponential	mam	Creates MAP representations of exponential inter-arrival time distributions with specified
map_feasblock	mam	Constructs feasible MAP representations when exact moment matching fails by adjusting
map_feastol	mam	Provides standard tolerance values for numerical feasibility checks in MAP algorithms
map_gamma	mam	Computes gamma parameter measuring autocorrelation decay rates in MAP processes
map_gamma2	mam	Computes largest non-unit eigenvalue of embedded DTMC for MAP correlation characterization
map_hyperexp	mam	Constructs MAP representations of two-phase hyperexponential renewal processes

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_idc	mam	Computes asymptotic index of dispersion for MAP counting processes, measuring long-term
map_infgen	mam	Computes infinitesimal generator matrix of underlying CTMC by combining MAP transition
map_isfeasible	mam	Provides convenient interface for MAP feasibility validation with configurable tolerance
map_joint	mam	Computes joint moments of MAP inter-arrival times for advanced statistical characterization
map_jointpdf_derivative	mam	Computes partial derivatives of MAP joint probability density functions at origin
map_kpc	mam	Computes Kronecker product composition of multiple MAPs for building complex arrival processes
map_kurt	mam	Computes kurtosis of MAP inter-arrival times measuring tail heaviness and distribution shape
map_lambda	mam	MAP arrival rate computation algorithms
map_largemap	mam	Provides size thresholds for determining when MAP algorithms should switch to
map_mark	mam	Creates Marked MAP (MMAP) representations by adding class labels to MAP arrivals
map_max	mam	Computes MAP representation of maximum inter-arrival times from independent MAP processes
map_mean	mam	MAP mean inter-arrival time computation algorithms
map_mixture	mam	Creates probabilistic mixtures of MAP processes with specified mixture probabilities
map_moment	mam	Computes raw moments of MAP inter-arrival times using matrix inversion techniques
map_normalize	mam	Sanitizes MAP matrices by ensuring non-negativity constraints and proper diagonal adjustments
map_pdf	mam	Computes PDF values for MAP inter-arrival times using matrix exponential techniques
map_pie	mam	Computes steady-state probability vector of embedded discrete-time Markov chain
map_piq	mam	Computes steady-state probability vector of underlying continuous-time Markov chain
map_pntiter	mam	Computes exact arrival probabilities using iterative numerical methods based on Neuts and Li
map_pntquad	mam	Computes MAP point process probabilities using ODE quadrature methods with Runge-Kutta integration
map_prob	mam	Computes equilibrium probability distribution of underlying CTMC for MAP analysis
map_rand	mam	Generates random MAP representations for testing, simulation, and statistical analysis
map_randn	mam	Generates random MAP samples with added numerical noise for robustness testing
map_renewal	mam	Creates renewal MAP by removing correlations to obtain memoryless arrival processes

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_sample	mam	Generates random samples from MAP distributions for simulation and empirical analysis
map_scale	mam	Rescales MAP inter-arrival time distributions to achieve specified mean values
map_scv	mam	Computes SCV of MAP inter-arrival times as normalized dispersion measure
map_skew	mam	Computes skewness of MAP inter-arrival times measuring asymmetry in distributions
map_stochcomp	mam	Performs state elimination through stochastic complementation while preserving MAP properties
map_sum	mam	Computes MAP representations of sums of identical MAP processes for load scaling
map_sumind	mam	Computes MAP representations of sums of independent MAP processes for modeling
map_super	mam	Creates superposition of MAP processes using Kronecker product techniques
map_timereverse	mam	Computes time-reversed MAP by adjusting transition rates based on stationary distributions
map_var	mam	MAP variance computation algorithms
map_varcount	mam	Computes variance of event counts in MAP processes over specified time intervals
maph2m_fit	mam	Fits MAPH(2,m) processes to match ordinary moments, class probabilities, and backward moments
maph2m_fit_mmap	mam	Fits MAPH(2,m) by approximating characteristics of input MMAP processes
maph2m_fit_multiclass	mam	Fits MAPH(2,m) models to multiclass characteristics with class-specific parameters
maph2m_fit_trace	mam	Fits MAPH(2,m) from empirical trace data for multiclass service time modeling
maph2m_fitc_approx	mam	Fits MAPH(2,m) using approximation methods for count statistics when exact solutions fail
maph2m_fitc_theoretical	mam	Fits MAPH(2,m) using theoretical count statistics for precise parameter estimation
mapqn_bnd_lr	mapqn	Implements general linear reduction methods for computing performance
mapqn_bnd_lr_mva	mapqn	Implements linear reduction bounds for MAP queueing networks using Mean
mapqn_bnd_lr_pf	mapqn	Implements linear reduction bounds specialized for product-form MAP
mapqn_bnd_qr	mapqn	Implements general quadratic reduction methods for computing performance
mapqn_bnd_qr_delay	mapqn	Implements quadratic reduction bounds for delay systems in MAP queueing
mapqn_bnd_qr_ld	mapqn	Implements quadratic reduction bounds for load-dependent MAP queueing

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mapqn_lpmodel	mapqn	Base class for representing MAP queueing network linear programming models
mapqn_parameters	mapqn	Defines the base parameter structure for MAP queueing network analysis
mapqn_parameters_factory	mapqn	Factory class for creating parameter objects for MAP queueing network
mapqn_qr_bounds_bas	mapqn	Implements Queue-Router bounds using the Balanced Asymptotic Scaling (BAS)
mapqn_qr_bounds_rsrd	mapqn	Implements Queue-Router (QR) bounds using the Randomized Simultaneous
mmap_backward_moment	mam	Computes backward moments of MMAP inter-arrival times for each marked class
mmap_compress	mam	Compresses MMAP using various approximation methods including mixture, matching,
mmap_count_idc	mam	Computes IDC values for each marked class in MMAP counting processes
mmap_count_lambda	mam	Computes arrival rate vectors for each marked class in MMAP processes
mmap_count_mcov	mam	Computes count covariance matrices between marked classes in MMAP processes
mmap_count_mean	mam	Computes mean count vectors for each marked class in MMAP counting processes
mmap_count_var	mam	Computes variance vectors for counting processes of each marked class in MMAP
mmap_cross_moment	mam	Computes cross-moment matrices between different marked classes in MMAP processes
mmap_embedded	mam	Computes embedded discrete-time Markov chain for MMAP processes
mmap_exponential	mam	Constructs MMAP with exponential inter-arrival distributions for each marked class
mmap_forward_moment	mam	Computes forward moments of MMAP inter-arrival times for each marked class
mmap_hide	mam	Hides specified arrival classes in MMAP processes by removing observable events
mmap_idc	mam	Computes asymptotic IDC for each marked class in MMAP as time approaches infinity
mmap_isfeasible	mam	Validates mathematical feasibility of MMAP representations including stochastic
mmap_issym	mam	Checks if an MMAP is symmetric
mmap_lambda	mam	MMAP arrival rate computation algorithms
mmap_maps	mam	Extracts individual MAP processes for each marked class from MMAP representations
mmap_mark	mam	Converts a Markovian Arrival Process with marked arrivals (MMAP) into a new MMAP with redefined classes based on a given probability matrix

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mmap_max	mam	Computes element-wise maximum of MMAP processes for synchronization analysis
mmap_mixture	mam	Creates probabilistic mixtures of MMAP processes with specified weights
mmap_mixture_fit	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given cross-moments
mmap_mixture_fit_mmap	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given moments
mmap_mixture_order2	mam	Creates a second-order MMAP mixture from a collection of MMAPs
mmap_modulate	mam	Modulates an MMAP by another MMAP, creating a compound arrival process
mmap_normalize	mam	Normalizes MMAP matrices to ensure feasibility and mathematical validity
mmap_pc	mam	Computes the proportion of counts (PC) for each type in a Markovian Arrival Process with marked arrivals (MMAP)
mmap_pie	mam	Computes steady-state probability vectors for each marked class in MMAP processes
mmap_rand	mam	Generates random MMAP representations for testing and simulation purposes
mmap_sample	mam	Generates random samples from MMAP distributions for each marked class
mmap_scale	mam	Rescales MMAP inter-arrival distributions to achieve specified mean values
mmap_shorten	mam	Converts an MMAP representation from M3A format to BUTools format
mmap_sigma	mam	Computes one-step class transition probabilities for a Marked Markovian Arrival Process (MMAP)
mmap_sigma2	mam	Computes two-step class transition probabilities for a Markovian Arrival Process (MMAP)
mmap_sum	mam	Computes superposition of MMAP processes creating independent multiclass arrival streams
mmap_super	mam	Combines multiple MMAP processes into superposed multiclass arrival streams
mmap_super_safe	mam	API function for mam operations
mmap_timereverse	mam	Computes the time-reversed version of a Markovian Arrival Process with marked arrivals (MMAP)
mmpp2_fit	mam	Fits MMPP(2) models to match specified moments and correlation characteristics
mmpp2_fit1	mam	Fits MMPP(2) models using simplified single-parameter approach for specific scenarios
mmpp2_fitc	mam	Fits MMPP(2) models using count statistics and index of dispersion criteria
mmpp2_fitc_approx	mam	Fits MMPP(2) using optimization-based approximation methods for count statistics

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mmpp_rand	mam	Generates random MMPP models with diagonal D1 matrices for testing and simulation
ms_additivesymmetricchisquared	measures	Additive symmetric chi-squared distance between two probability distributions
ms_adtest	measures	Implements the Anderson-Darling test for assessing whether a sample comes from
ms_avgl1linfty	measures	Average L1 L-infinity distance between two probability distributions
ms_bhattacharyya	measures	Computes the Bhattacharyya distance measuring the similarity between probability
ms_canberra	measures	Canberra distance between two probability distributions
ms_chebyshev	measures	Chebyshev Distance for Probability Distributions
ms_chisquared	measures	Squared chi-squared distance between two probability distributions
ms_cityblock	measures	Implements the City block distance between probability distributions
ms_clark	measures	Clark distance between two probability distributions
ms_condentropy	measures	Computes conditional entropy measuring the remaining uncertainty
ms_cramer_von_mises	measures	Implements the Cramer-von Mises test statistic for comparing two empirical distributions
ms_cosine	measures	Computes cosine distance (1 - cosine similarity) measuring the angle between two
ms_czekanowski	measures	Czekanowski distance between two probability distributions
ms_dice	measures	Dice distance between two probability distributions
ms_divergence	measures	Divergence distance between two probability distributions
ms_entropy	measures	Implements Shannon entropy for discrete random variables
ms_euclidean	measures	Implements the standard Euclidean distance between probability distributions
ms_fidelity	measures	Fidelity distance between two probability distributions
ms_gower	measures	Gower distance between two probability distributions
ms_harmonicmean	measures	Harmonic mean distance between two probability distributions
ms_hellinger	measures	Computes the Hellinger distance measuring dissimilarity between probability distributions
ms_intersection	measures	Intersection distance between two probability distributions
ms_jaccard	measures	Jaccard distance between two probability distributions
ms_jeffreys	measures	Jeffreys divergence between two probability distributions
ms_jensendifference	measures	Jensen difference divergence between two probability distributions
ms_jensenshannon	measures	Implements Jensen-Shannon divergence, a symmetric and bounded version of
ms_jointentropy	measures	Computes joint entropy measuring the uncertainty of joint distributions
ms_kdivergence	measures	K-divergence between two probability distributions

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_kolmogorov_smirnov	measures	Implements the Kolmogorov-Smirnov test for determining if a sample follows
ms_kuiper	measures	Implements the Kuiper test statistic, a rotation-invariant variant of the Kolmogorov-Smirnov
ms_kulczynskid	measures	Kulczynski d distance between two probability distributions
ms_kulczynskis	measures	Kulczynski s distance between two probability distributions
ms_kullbackleibler	measures	Implements the Kullback-Leibler divergence measuring distribution differences
ms_kumarhassebrook	measures	Kumar-Hassebrook distance between two probability distributions
ms_kumarjohnson	measures	Kumar-Johnson distance between two probability distributions
ms_lorentzian	measures	Lorentzian distance between two probability distributions
ms_matusita	measures	Matusita distance between two probability distributions
ms_minkowski	measures	Minkowski Distance for Probability Distributions
ms_motyka	measures	Motyka distance between two probability distributions
ms_mutinfo	measures	Computes mutual information measuring the amount of shared information
ms_neymanchisquared	measures	Neyman chi-squared distance between two probability distributions
ms_nmi	measures	Computes normalized mutual information providing a scale-invariant measure
ms_nvi	measures	Computes normalized variation information measuring the normalized distance
ms_pearsonchisquared	measures	Pearson chi-squared distance between two probability distributions
ms_probsymmchisquared	measures	Probabilistic symmetry chi-squared distance between two probability distributions
ms_relatentropy	measures	Computes relative entropy measuring the information difference
ms_product	measures	Product distance between two probability distributions
ms_ruzicka	measures	Ruzicka distance between two probability distributions
ms_soergel	measures	Soergel distance between two probability distributions
ms_sorensen	measures	Sorensen distance between two probability distributions
ms_squaredchord	measures	Squared chord distance between two probability distributions
ms_squaredeuclidean	measures	Squared Euclidean distance between two probability distributions
ms_taneja	measures	Taneja distance between two probability distributions
ms_tanimoto	measures	Tanimoto distance between two probability distributions
ms_topsoe	measures	Topsoe distance between two probability distributions
ms_wasserstein	measures	Implements the Wasserstein distance measuring the minimum cost to transform
ms_wavehegdes	measures	Wave-Hedges distance between two probability distributions
mtrace_backward_moment	trace	Computes backward moments of a multi-class trace
mtrace_bootstrap	trace	Implements bootstrap resampling methods for multi-class empirical trace data

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mtrace_count	trace	Computes count statistics from a multi-class trace over specified time windows
mtrace_cov	trace	Computes the covariance matrix for multi-type traces
mtrace_cross_moment	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_forward_moment	trace	Computes the forward moments of a marked trace
mtrace_iat2counts	trace	Computes the per-class counting processes of T, i.e., the counts after
mtrace_joint	trace	Given a multi-class trace, computes the empirical class-dependent joint
mtrace_mean	trace	Computes per-class means for multi-class empirical trace data. Enables separate
mtrace_merge	trace	Merges two traces in a single marked (multiclass) trace
mtrace_moment	trace	Computes empirical class-dependent statistical moments for multi-class trace data
mtrace_moment_simple	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_pc	trace	Computes the probabilities of arrival for each class
mtrace_sigma	trace	Computes the empirical probability of observing a specific 2-element
mtrace_sigma2	trace	Computes the empirical probability of observing a specific 3-element
mtrace_split	trace	Given a multi-class trace with inter-arrivals T and labels L,
mtrace_summary	trace	Computes summary statistics for multiple trace analysis, providing
npfqn_nonexp_approx	npfqn	Implements approximation methods for non-product-form queueing networks
npfqn_traffic_merge	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_merge_cs	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_split_cs	npfqn	Implements traffic splitting algorithms for non-product-form queueing networks
pfqn_ab	pfqn.ld	Implements the Akyildiz-Bolch linearizer method for analyzing closed product-form queueing networks
pfqn_aql	pfqn.mva	Implements the Aggregate Queue Length (AQL) approximation method for analyzing closed
pfqn_bs	pfqn.mva	Implements the classic Bard-Schweitzer approximate MVA algorithm for closed queueing networks
pfqn_ca	pfqn.nc	Convolution Algorithm for Product-Form Networks
pfqn_cdfun	pfqn.ld	Provides functionality to evaluate class-dependent scaling functions in load-dependent queueing networks
pfqn_comomrm	pfqn.nc	Implements the Convolution Method of Moments specialized for repairman queueing models
pfqn_comomrm_ld	pfqn.ld	Implements the Convolution Method of Moments (COMOM) for computing normalizing constants in

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_conwayms	pfqn.mva	Implements the Conway-Maxwell approximation method for analyzing closed queueing networks
pfqn_cub	pfqn.nc	Implements the cubature (multi-dimensional integration) approach for computing normalizing
pfqn_egflinearizer	pfqn.mva	Implements the Extended General-Form linearizer approximation for closed queueing networks
pfqn_fnc	pfqn.ld	Computes scaling factors for load-dependent functional servers in product-form queueing networks
pfqn_gflinearizer	pfqn.mva	Implements the general-form linearizer approximation for closed queueing networks with
pfqn_gld	pfqn.ld	Implements the generalized convolution algorithm for computing normalizing constants in
pfqn_gld_complex	pfqn.ld	Extends the generalized load-dependent convolution algorithm to handle complex-valued
pfqn_gldsingle	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_gldsingle_complex	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_kt	pfqn.nc	Implements the Knessl-Tier asymptotic expansion using the ray method for computing
pfqn_le	pfqn.nc	Implements the Laguerre expansion approach for computing normalizing constants in
pfqn_le_fpi	pfqn.nc	Implements the fixed-point iteration algorithm used in the Laguerre expansion method
pfqn_le_fpiz	pfqn.nc	Implements the fixed-point iteration algorithm for the Laguerre expansion method
pfqn_le_hessian	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for second-order
pfqn_le_hessianz	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for closed queueing
pfqn_linearizer	pfqn.mva	Linearizer Approximate MVA for Product-Form Networks
pfqn_linearizerms	pfqn.mva	Implements the multi-server version of Krzesinski's linearizer approximation for closed
pfqn_linearizermx	pfqn.mva	Implements linearizer-based approximation methods for mixed queueing networks with
pfqn_linearizerpp	pfqn.mva	Implements the Linearizer++ algorithm for closed queueing networks with enhanced accuracy
pfqn_lldfun	pfqn.ld	Evaluates limited load-dependent (LLD) scaling functions using spline interpolation for
pfqn_ls	pfqn.nc	Implements the logistic sampling approach for computing normalizing constants in
pfqn_mci	pfqn.nc	Implements Monte Carlo integration approaches including Importance Monte Carlo Integration
pfqn_mmint2	pfqn.nc	Implements numerical integration for computing normalizing constants in multi-class

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_mmint2_gausslegendre	pfqn.nc	Implements Gauss-Legendre quadrature integration for computing normalizing constants
pfqn_mmsample2	pfqn.nc	Implements importance sampling for computing normalizing constants in multi-class
pfqn_mom	pfqn.nc	Implements the Method of Moments using exact arithmetic with BigFraction for computing
pfqn_mu_ms	pfqn.ld	Computes load-dependent scaling factors for multi-server queueing stations with finite
pfqn_mushift	pfqn.ld	Provides utility function for shifting load-dependent scaling vectors by one position,
pfqn_mva	pfqn.mva	Mean Value Analysis for Product-Form Queueing Networks
pfqn_mvald	pfqn.ld	Load-Dependent Mean Value Analysis
pfqn_mvaldms	pfqn.ld	Provides wrapper functionality for load-dependent Mean Value Analysis with automatic
pfqn_mvaldmx	pfqn.ld	Implements Mean Value Analysis for mixed queueing networks with both open and closed classes
pfqn_mvaldmx_ec	pfqn.ld	Provides auxiliary functionality for computing EC terms used in load-dependent Mean Value
pfqn_mvams	pfqn.mva	Provides comprehensive MVA solution for mixed queueing networks with multi-server stations
pfqn_mvamx	pfqn.mva	Implements MVA for mixed networks containing both open and closed classes without multi-server
pfqn_nc	pfqn.nc	Normalizing Constant Methods for Product-Form Networks
pfqn_nc_sanitize	pfqn.nc	Sanitizes and preprocesses parameters for product-form queueing network models to
pfqn_nca	pfqn.nc	Implements the Normalizing Constant Approximation method for single-class closed
pfqn_ncld	pfqn.ld	Provides the main entry point for computing normalizing constants in load-dependent
pfqn_nrl	pfqn.nc	Implements the Normal Random Lattice approach for computing normalizing constants
pfqn_nrp	pfqn.nc	Implements the Normal Random Permutation approach for computing normalizing constants
pfqn_panacea	pfqn.nc	Implements the PANACEA approximation method for computing normalizing constants in
pfqn_pff_delay	pfqn.nc	Computes the product-form factor for delay stations in closed queueing networks
pfqn_procomom2	pfqn.ld	Implements the probabilistic class-oriented method of moments for analyzing
pfqn_propfair	pfqn.nc	Implements the proportionally fair allocation method using convex optimization
pfqn_qzgblow	pfqn.mva	Computes the lower Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_qzgbup	pfqn.mva	Computes the upper Geometric Bound (GB) for queue lengths in closed single-class queueing

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_rd	pfqn.nc	Implements the Random Discretization approach for computing normalizing constants in
pfqn_recal	pfqn.nc	Implements the RECAL (Recursive Calculation) algorithm for computing normalizing constants
pfqn_schmidt	pfqn.ld	Schmidt method for load-dependent MVA with multi-server stations
pfqn_sqni	pfqn.mva	Implements the Single Queue Network Interpolation method for analyzing multi-class closed
pfqn_stdf	pfqn.nc	Implements McKenna's 1987 method for computing sojourn time distributions at
pfqn_stdf_heur	pfqn.nc	Implements a heuristic variant of McKenna's 1987 method for computing sojourn time
pfqn_xia	pfqn.ld	Implements Xia's asymptotic approximation method for computing normalizing constants
pfqn_xzabalow	pfqn.mva	Computes the lower ABA bound for throughput in closed single-class queueing networks
pfqn_xzabaup	pfqn.mva	Computes the upper ABA bound for throughput in closed single-class queueing networks
pfqn_xzgsblow	pfqn.mva	Computes the lower GSB for throughput in closed single-class queueing networks using
pfqn_xzgsbup	pfqn.mva	Computes the upper GSB for throughput in closed single-class queueing networks using
ph_reindex	mam	Reindexes phase-type distribution maps for network models using integer station and class indices
polling_qsys_1limited	polling	Implements analysis algorithms for 1-limited polling systems where the
polling_qsys_exhaustive	polling	Implements analysis algorithms for exhaustive polling systems where the
polling_qsys_gated	polling	Implements analysis algorithms for gated polling systems where the server
qbd_bmapbmap1	mam	Analyzes batch arrival and service systems using QBD matrix methods
qbd_mapmap1	mam	Analyzes MAP/MAP/1 queueing systems using QBD matrix analytic methods
qbd_r	mam	QBD R-matrix computation algorithms
qbd_r_logred	mam	Computes QBD R-matrix using logarithmic reduction method for numerical stability
qbd_raprap1	mam	Analyzes RAP/RAP/1 queueing systems using QBD methods with rational arrival processes
qbd_rg	mam	Computes fundamental R and G matrices for QBD analysis of MAP/MAP/1 queues
qbd_setupdelayoff	mam	Analyzes queueing systems with server setup delays and switch-off mechanisms
qsys_gg1	qsys	Provides comprehensive analysis of G/G/1 queues with general arrival and service processes

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
qsys_gig1_approx_allencunneen	qsys	Implements the widely-used Allen-Cunneen approximation for general G/G/1 queueing
qsys_gig1_approx_gelenbe	qsys	G/G/1 queue approximation using Gelenbe's method
qsys_gig1_approx_heyman	qsys	Analyzes a G/G/1 queueing system using Heyman's approximation
qsys_gig1_approx_kimura	qsys	G/G/1 queue approximation using Kimura's method
qsys_gig1_approx_klb	qsys	Analyzes a G/G/1 queueing system using the Kramer-Langenbach-Belz (KLB) approximation
qsys_gig1_approx_kobayashi	qsys	Analyzes a G/G/1 queueing system using Kobayashi's approximation
qsys_gig1_approx_marchal	qsys	Analyzes a G/G/1 queueing system using Marchal's approximation
qsys_gig1_approx_myskja	qsys	G/G/1 queue approximation using Myskja's method
qsys_gig1_approx_myskja2	qsys	G/G/1 queue approximation using enhanced Myskja's method
qsys_gig1_lbnd	qsys	G/G/1 queue lower bounds
qsys_gig1_ubnd_kingman	qsys	Calculates an upper bound on the waiting time for a G/G/1 system using Kingman's formula
qsys_gigk_approx	qsys	Analyzes a G/G/k queueing system using an approximation method
qsys_gigk_approx_cosmetatos	qsys	G/G/k queue approximation using Cosmetatos method
qsys_gigk_approx_kingman	qsys	Analyzes a G/G/k queueing system using Kingman's approximation
qsys_gigk_approx_whitt	qsys	G/G/k queue approximation using Whitt's method
qsys_gm1	qsys	G/M/1 Queueing System Analysis
qsys_mg1	qsys	Implements the Pollaczek-Khinchine formula for M/G/1 queues with Poisson arrivals
qsys_mg1k_loss	qsys	M/G/1/K loss probability calculation
qsys_mg1k_loss_mgs	qsys	M/G/1/K loss probability using MacGregor Smith approximation
qsys_mginf	qsys	M/G/inf queue analysis (infinite servers)
qsys_mm1	qsys	Implements exact analytical solutions for the M/M/1 queue (Poisson arrivals, exponential
qsys_mm1k_loss	qsys	M/M/1/K loss probability calculation
qsys_mmk	qsys	Implements exact analytical solutions for M/M/k queues with Poisson arrivals,
randp	mam	Provides random value selection based on relative probability distributions
rl_env	rl	Provides a reinforcement learning environment interface for queueing networks,
rl_env_general	rl	Provides a general reinforcement learning environment for queueing networks
rl_td_agent	rl	Implements a temporal difference learning agent for queueing network control
rl_td_agent_general	rl	Implements a general-purpose temporal difference learning agent for queueing

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_deaggregate_chain_results	sn	Calculate class-based performance metrics for a queueing network based on performance measures of its chains
sn_get_arv_r_from_tput	sn	Calculates the average arrival rates at each station from the network throughputs
sn_get_demands_chain	sn	Calculate new queueing network parameters after aggregating classes into chains
sn_get_node_arv_r_from_tput	sn	API function for sn operations
sn_get_node_tput_from_tput	sn	APIs to process NetworkStruct objects
sn_get_product_form_chain_params	sn	Calculate the parameters at class and chain level for a queueing network model
sn_get_product_form_params	sn	Extracts essential parameters (service demands, populations, visit ratios) from
sn_get_residt_from_respt	sn	Calculates the residence times at each station from the response times
sn_get_state_aggr	sn	Aggregates the state of the network
sn_has_class_switching	sn	Checks if the network uses class-switching
sn_has_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_dps	sn	Stochastic network HasDPS algorithms
sn_has_dps_prio	sn	Stochastic network HasDPSPRIO algorithms
sn_has_fcfs	sn	Identifies queueing networks using First-Come-First-Served scheduling disciplines
sn_has_fork_join	sn	Checks if the network uses fork and/or join nodes
sn_has_fractional_populations	sn	Checks if the network has closed classes with non-integer populations
sn_has_gps	sn	Stochastic network HasGPS algorithms
sn_has_gps_prio	sn	Stochastic network HasGPSPRIO algorithms
sn_has_hol	sn	Stochastic network HasHOL algorithms
sn_has_homogeneous_scheduling	sn	Checks if the network uses an identical scheduling strategy at every station
sn_has_inf	sn	Stochastic network HasINF algorithms
sn_has_lcfs	sn	Stochastic network HasLCFS algorithms
sn_has_lcfs_pr	sn	Stochastic network HasLCFSPR algorithms
sn_has_lept	sn	Stochastic network HasLEPT algorithms
sn_has_ljf	sn	Stochastic network HasLJF algorithms
sn_has_load_dependence	sn	Checks if the network has a station with load-dependent service process
sn_has_mixed_classes	sn	Checks if the network has both open and closed classes
sn_has_multi_chain	sn	Stochastic network HasMultiChain algorithms
sn_has_multi_class	sn	Identifies queueing networks with multiple job classes, which require specialized
sn_has_multi_class_fcfs	sn	API function for sn operations
sn_has_multi_class_heter_exp_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_class_heter_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_server	sn	Stochastic network HasMultiServer algorithms

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_has_multiple_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_open_classes	sn	Checks if the network has one or more open classes
sn_has_polling	sn	Stochastic network HasPolling algorithms
sn_has_priorities	sn	Checks if the network uses class priorities
sn_has_product_form	sn	Determines if a queueing network has a known product-form solution by validating
sn_has_product_form_not_het_fcfs	sn	Checks if the network satisfies product-form assumptions (does not have heterogeneous FCFS)
sn_has_ps	sn	Stochastic network HasPS algorithms
sn_has_ps_prio	sn	Stochastic network HasPSPRIO algorithms
sn_has_sept	sn	Stochastic network HasSEPT algorithms
sn_has_single_chain	sn	Stochastic network HasSingleChain algorithms
sn_has_single_class	sn	Stochastic network HasSingleClass algorithms
sn_has_siro	sn	Stochastic network HasSIRO algorithms
sn_has_sjf	sn	Stochastic network HasSJF algorithms
sn_is_closed_model	sn	Identifies closed queueing network models with finite job populations and no external
sn_is_mixed_model	sn	Checks if the network is a mixed model
sn_is_open_model	sn	Identifies open queueing network models with external arrivals and infinite
sn_is_population_model	sn	Checks if the model is a population model (only specific scheduling strategies without priorities or fork-join)
sn_is_state_valid	sn	Stochastic Network State Validation Utility
sn_print	sn	Prints comprehensive information about a NetworkStruct
sn_print_routing_matrix	sn	Prints the routing matrix of the network, optionally for a specific job class
sn_refresh_visits	sn	Stochastic Network Visit Ratio Calculator
sn_rtnodes_to_rtorig	sn	Converts routing matrices from nodes to original format, specifically handling class switching nodes
trace_mean	trace	Computes the arithmetic mean of empirical trace data. Fundamental statistical
trace_skew	trace	Computes the skewness of the trace data using Apache Commons Math
trace_var	trace	Computes sample variance and related statistics for empirical trace data
wkflow_analyzer	wkflow	Provides comprehensive workflow analysis capabilities including pattern
wkflow_auto_integration	wkflow	Provides automatic integration capabilities for workflow analysis with
wkflow_branch_detector	wkflow	Implements algorithms for detecting branching patterns in workflow traces
wkflow_loop_detector	wkflow	Implements algorithms for detecting loop and iterative patterns in workflow
wkflow_parallel_detector	wkflow	Implements algorithms for detecting parallel execution patterns in workflow

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`wkflow_pattern_updater`	wkflow	Implements dynamic pattern updating algorithms for workflow analysis
`wkflow_sequence_detector`	wkflow	Implements algorithms for detecting sequential patterns in workflow traces

Index