# LINE: Performance and Reliability Analysis Engine

## User manual

Last revision: August 10, 2019

# Contents

# Chapter 1

# Introduction

## 1.1 What is LINE?

LINE is an engine for system performance and reliability evaluation based on queueing theory and stochastic modeling. The goal of the tool is to simplify the computation of performance and reliability metrics in 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 for performance and reliability metrics using either numerical algorithms or simulation.

A key feature of LINE is that the engine decouples the model description from the solvers used for its solution. That is, the engine 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 (`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 (MATLAB script format)*. This is a MATLAB-based 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 in MATLAB 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.
- *Performance Model Interchange Format (PMIF XML format)*. LINE is able to import and solve closed queueing network models specified using PMIF v1.0.

## 1.2 Obtaining the latest release

This document contains the user manual for LINE version 2.0.0, which can be obtained from:

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

LINE 2.0.0 has been tested on MATLAB R2018a and later releases and requires the *Statistics and Machine Learning Toolbox*. Some advanced features also require the *Parallel Computing Toolbox*. If you are interested to obtain LINE in another format, please contact us via the discussion forum (https://sourceforge.net/p/line-solver/discussion/help/).

## 1.3 References

To cite LINE, we recommend to reference:

- G. Casale. "Automated Multi-paradigm Analysis of Extended and Layered Queueing Models with LINE", in *Proc. of ACM/SPEC 2019*, ACM Press, Apr 2019. *This paper introduces* LINE *2.0.0.*

The following papers discuss some applications of LINE and earlier versions of the tool:

- J. F. Pérez and G. Casale. "LINE: Evaluating Software Applications in Unreliable Environments", in *IEEE Transactions on Reliability*, Volume 66, Issue 3, pages 837-853, Feb 2017. *This paper introduces the core algorithms behind* LINE *1.0.0.*

- C. Li and G. Casale. "Performance-Aware Refactoring of Cloud-based Big Data Applications", in Proceedings of 10th IEEE/ACM International Conference on Utility and Cloud Computing, 2017. *This paper uses* LINE *to model stream processing systems*.
- D. J. Dubois, G. Casale. "OptiSpot: minimizing application deployment cost using spot cloud resources", in *Cluster Computing*, Volume 19, Issue 2, pages 893-909, 2016. *This paper uses* LINE *to determine bidding costs in spot VMs*.
- R. Osman, J. F. Pérez, and G. Casale. "Quantifying the Impact of Replication on the Quality-of-Service in Cloud Databases'. Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 286-297, 2016. *This paper uses* LINE *to model the Amazon RDS database*.
- C. Müller, P. Rygielski, S. Spinner, and S. Kounev. Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation, Electr. Notes Theor. Comput. Sci, 327, 71–91, 2016. *This paper uses* LINE *to analyze Descartes models used in software engineering*.
- J. F. Pérez and G. Casale. "Assessing SLA compliance from Palladio component models," in Proceedings of the 2nd Workshop on Management of resources and services in Cloud and Sky computing (MICAS), IEEE Press, 2013. *This paper uses* LINE *to analyze Palladio component models used in model-driven software engineering*.

## 1.4 Contact and credits

Project coordinator and maintainer:

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 following file for detailed credits:
AUTHORS: https://github.com/line-solver/line/blob/master/AUTHORS

## 1.5 Copyright and license

License files of third-party libraries are placed under the `lib/thirdparty/` directory.

## 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

This is the fastest way to get started with LINE:

1. Obtain the latest release:

   - Stable release (zip file): https://sourceforge.net/projects/line-solver/files/latest/download
   - Development release (git): https://sourceforge.net/p/line-solver/code/ci/master/tree/

   Ensure that the files are decompressed (or checked out) in the installation folder.

2. Before running LINE you will always need to add the installation folder and its subfolders to the MATLAB path. In order to do so, start MATLAB and change the active directory to the installation folder, then run

```
addpath(genpath(pwd))
```

3. LINE is now ready to use. For example, you can run the demonstrators using

```
allExamples
```

For optimal visualization of the solver results we recommend running the following commands before invoking LINE

```
format compact
warning OFF BACKTRACE
```

### 2.1.1 Software requirements

Certain features of LINE depend on external tools and libraries. The minimal requirement is:

- MATLAB: version 2018a or later, with the *Statistics and Machine Learning Toolbox*.

The following libraries are automatically downloaded or shipped with LINE:

- Java Modelling Tools (http://jmt.sf.net): version 1.0.4 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 0.3.2 or later. This release is already included under the $lib/$ subfolder.
- M3A Toolbox (https://github.com/Imperial-AESOP/M3A): version 1.0.0. This release is already included under the $lib/$ subfolder.
- BuTools (http://webspn.hit.bme.hu/~telek/tools/butools/butools2.zip): version 2.0 or later. This release is already included under the $lib/$ subfolder.
- SMCSolver (https://win.uantwerpen.be/~vanhoudt/tools/QBDfiles.zip): This release is already included under the $lib/$ subfolder.

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

- LQNS (http://www.sce.carleton.ca/rads/lqns/): version 6.0 or later. System paths need to be configured such that the `lqns` and `lqnsim` solvers need are available on the command line (i.e., can be invoked from MATLAB via the `dos` or `unix` commands without need to specify the paths of the executables).
- MATLAB *Parallel Computing Toolbox*. This is required to use the parallel simulation capabilities of the SSA solver.

### 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:

- An online wiki version of this manual: https://github.com/line-solver/LINE/wiki.
- APIs reference: http://line-solver.sourceforge.net/api/index.html.

### 2.1.3 Getting help

For discussions, bug reports, new feature requests, please create a thread on one of the following Sourceforge boards:

- General discussion: https://sourceforge.net/p/line-solver/discussion/help/
- Bugs and issues: https://sourceforge.net/p/line-solver/tickets/
- Feature requests: https://sourceforge.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 `gettingstarted/` folder.

Systems can be described in LINE using one of the available classes of stochastic models:

- `Network` models are extended queueing networks. Typical instances are open, closed and mixed queueing networks, 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, 20] or in tutorials [1, 19].
- `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 [28].

The goal of the remainder of this chapter is to provide simple examples that explain the basics on how these models can be analyzed in LINE. More advanced forms of evaluation, such as probabilistic or transient analyses, are discussed in later chapters. Additional examples are supplied under the `examples/` and `gallery/` folders, the latter is discussed in the next subsection.

### 2.2.1 Model gallery

`Line` includes a collection of 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 using JMT a tandem $M/M/1 \rightarrow M/M/1$ network we may run

```
>> SolverMVA(gallery_mm1_tandem).getAvgRespTTable
MVA analysis (method: default) completed in 0.054887 seconds.
ans =
  2x3 table
    Station       Class      RespT
```

```
    ________    _________    _____
    'Queue1'    'myClass'      9
    'Queue2'    'myClass'      9
```

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.

### 2.2.2 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-served (FCFS) order, and then leave after service completion. Arrival and service times are assumed to be independent and exponentially distributed random variables.

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

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

```
model = Network('M/M/1');
%% Block 1: nodes
source = Source(model, 'mySource');
queue = Queue(model, 'myQueue', SchedStrategy.FCFS);
sink = Sink(model, 'mySink');
%% Block 2: classes
oclass = OpenClass(model, 'myClass');
source.setArrival(oclass, Exp(1));
queue.setService(oclass, Exp(2));
%% Block 3: topology
model.link(Network.serialRouting(source,queue,sink));
%% Block 4: solution
AvgTable = SolverJMT(model,'seed',23000).getAvgTable
```

In the example, `source` and `sink` are arrival and departure points of jobs; `queue` is a queueing station with FCFS scheduling; `oclass` defines an open class of jobs that arrive, get served, and leave the system; `Exp(x)` defines an exponential distribution with rate `x`; finally, the `getAvgTable` 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 per visit to the station (`RespT`); the mean throughput of departing jobs (`Tput`)

```
AvgTable =
  2x6 table
     Station        Class        QLen       Util       RespT      Tput
    ___________    _________    _______    ________    _______    _______
    'mySource'     'myClass'          0           0          0    0.99894
    'myQueue'      'myClass'     0.9555     0.48736    0.95429    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.

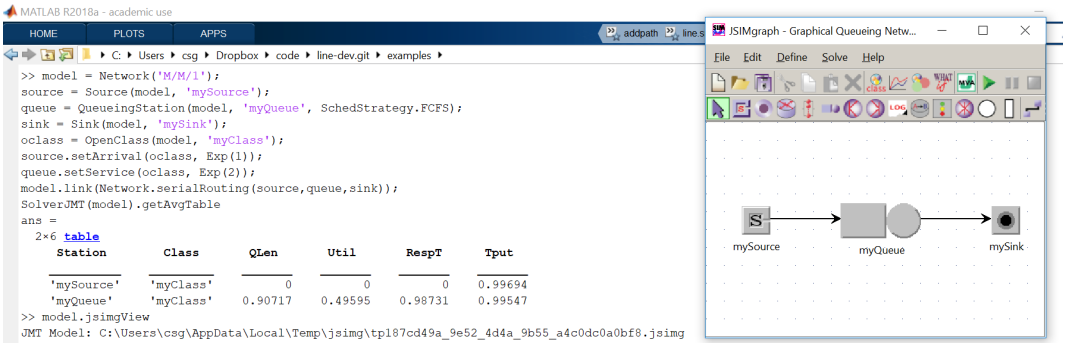


Figure 2.1: M/M/1 example in JSIMgraph

LINE also offers shortcuts to accelerate the definition of basic models. For example, an M/M/1 with arrival rate 1 and service rate 2 can be readily instantiated as

```
model = Network.tandemFcfs(1,2)
```

A pre-defined gallery of classic models is also available, for example

```
model = gallery_mm1
```

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

### 2.2.3 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 MATLAB working directory to the `examples/` folder. Then we specify the node block

```
model = Network('M/G/1');
source = Source(model,'Source');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);
sink = Sink(model,'Sink');
```

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

```
jobclass1 = OpenClass(model, 'Class1');
jobclass2 = OpenClass(model, 'Class2');

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

queue.setService(jobclass1, Erlang.fitMeanAndSCV(1, 1/3));
queue.setService(jobclass2, Replayer('example_trace.txt'));
```

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

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

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

```
P = model.initRoutingMatrix();
P{jobclass1} = Network.serialRouting(source,queue,sink);
P{jobclass2} = Network.serialRouting(source,queue,sink);
model.link(P);
```

and solve the model with JMT

```
>>jmtAvgTable = SolverJMT(model,'seed',23000).getAvgTable
jmtAvgTable =
```

```
  4x6 table
    Station      Class       QLen        Util       RespT      Tput
    _________    _________   ________    _________  ________   _______
    'Source'     'Class1'         0           0          0     0.5022
    'Source'     'Class2'         0           0          0    0.49999
    'Queue'      'Class1'   0.92002     0.51294     1.7724     0.5071
    'Queue'      'Class2'   0.44506    0.051639    0.85211    0.49996
```

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('example_trace.txt').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

```
>> ctmcAvgTable2 = SolverCTMC(model,'cutoff',2,'verbose',true).getAvgTable
State space size: 46 states.
CTMC analysis completed in 0.096734 sec
ctmcAvgTable2 =
  4x6 table
    Station      Class       QLen        Util       RespT      Tput
    _________    _________   ________    _________  ________   _______
    'Source'     'Class1'         0           0          0    0.44948
    'Source'     'Class2'         0           0          0    0.48424
    'Queue'      'Class1'   0.56734     0.44948     1.2863    0.44107
    'Queue'      'Class2'   0.24455    0.048942    0.51396    0.47583
```

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

```
>> ctmcAvgTable4 = SolverCTMC(model,'cutoff',4,'verbose',true).getAvgTable
State space size: 626 states.
CTMC analysis completed in 1.051784 sec
ctmcAvgTable4 =
  4x6 table
    Station      Class       QLen        Util       RespT      Tput
    _________    _________   ________    _________  ________   _______
    'Source'     'Class1'         0           0          0    0.49215
    'Source'     'Class2'         0           0          0    0.49626
    'Queue'      'Class1'    0.7958     0.49215     1.6187    0.49162
    'Queue'      'Class2'   0.37558    0.050157    0.75763    0.49573
```

To gain more accuracy, we could either keep increasing the cutoff value or, if we wish to compute an exact solution, we may call the matrix-analytic method (MAM) solver instead. MAM uses the repetitive structure of the CTMC to exactly analyze open systems with an infinite state space

```
>> mamAvgTable = SolverMAM(model).getAvgTable
mamAvgTable =
  4x6 table
    Station      Class        QLen        Util       RespT      Tput
    _________    _________    ________    ________    _______    ____
    'Source'     'Class1'          0           0          0       0.5
    'Source'     'Class2'          0           0          0       0.5
    'Queue'      'Class1'    0.87646         0.5     1.7529       0.5
    'Queue'      'Class2'      0.427    0.050536    0.85399       0.5
```

The current `MAM` implementation is primarily constructed on top of the BuTools solver [18] and the SMC solver [3].

### 2.2.4 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 operates as normal for an exponential distributed time with rate $4.0$ failed 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:

```
S=2; N=3;
model = Network('MIP');
%% Block 1: nodes
delay = Delay(model,'WorkingState');
queue = Queue(model, 'RepairQueue', SchedStrategy.FCFS);
queue.setNumberOfServers(S);
%% Block 2: classes
cclass = ClosedClass(model, 'Machines', N, delay);
delay.setService(cclass, Exp(0.5));
queue.setService(cclass, Exp(4.0));
%% Block 3: topology
model.link(Network.serialRouting(delay,queue));
%% Block 4: solution
solver = SolverCTMC(model);
ctmcAvgTable = solver.getAvgTable()
```

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

```
ctmcAvgTable =
  2x6 table
      Station          Class         QLen       Util       RespT      Tput
    _____________    ___________    _______    _______    _______    _______
    'WorkingState'   'Machines'      2.6648     2.6648          2     1.3324
    'RepairQueue'    'Machines'     0.33516    0.16655    0.25154     1.3324
```

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.

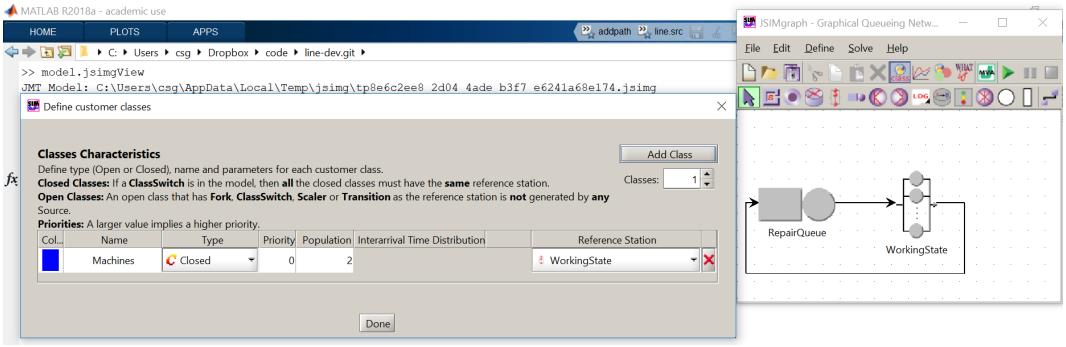


Figure 2.2: Machine interference model in JSIMgraph

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

```
StateSpace = solver.getStateSpace()
InfGen = full(solver.getGenerator())
```

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

```
StateSpace =
     0     1     2
     1     0     2
     2     0     1
     3     0     0
InfGen =
   -8.0000    8.0000         0         0
    0.5000   -8.5000    8.0000         0
         0    1.0000   -5.0000    4.0000
         0         0    1.5000   -1.5000
```

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`.

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 `InfGen(1,2)=8.0`, which sums the completion rates at the queue for each server, and where indexes 1 and 2 are the rows in `StateSpace` associated to the two states.

### 2.2.5 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:

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

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

```
oclass = OpenClass(model, 'Class1');
source.setArrival(oclass, Exp(1));
queue1.setService(oclass, Exp(2));
queue2.setService(oclass, Exp(2));
```

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([source, lb;
                lb,     queue1;
                lb,     queue2;
                queue1, sink;
                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 $0.66s$.

```
>> jmtAvgTable = SolverJMT(model,'seed',23000).getAvgTable
jmtAvgTable =
  3x6 table
    Station       Class        QLen        Util        RespT       Tput
    _________    _________    ________    ________    ________    _______
    'Source'     'Class1'          0           0           0      1.0135
    'Queue1'     'Class1'    0.31612     0.24682     0.65411       0.501
    'Queue2'     'Class1'    0.33403     0.25076     0.68406     0.50413
```

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(oclass, RoutingStrategy.RR);
```

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

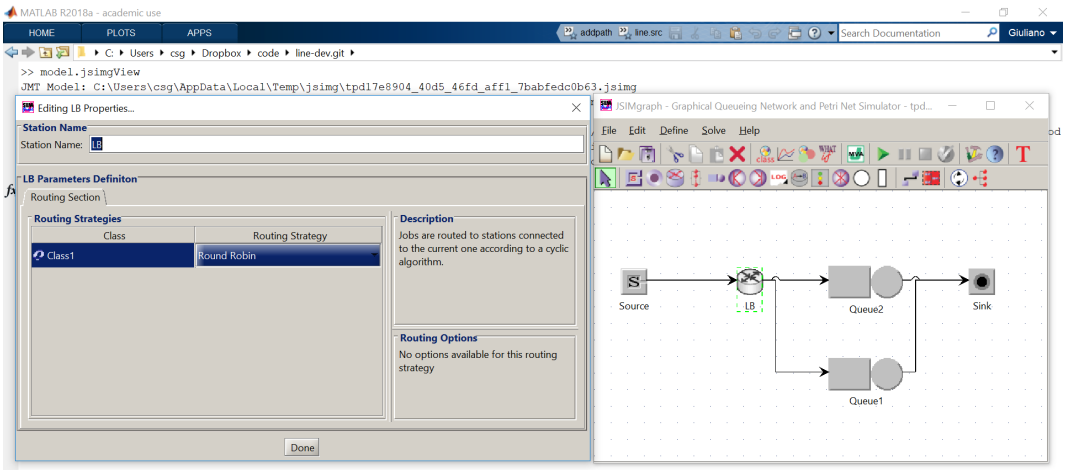


Figure 2.3: Load-balancing model

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

```
>> jmtAvgTableRR = SolverJMT(model,'seed',23000).getAvgTable
jmtAvgTableRR =
  3x6 table
    Station       Class        QLen        Util        RespT       Tput
```

```
    'Source'    'Class1'         0         0         0    1.0089
    'Queue1'    'Class1'   0.30429   0.26118   0.58482   0.50526
    'Queue2'    'Class1'   0.29282   0.24397   0.57293   0.50526
```

### 2.2.6 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 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.

```
model = Network('RL');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);

K = 3; N = [1,0,0];
for k=1:K
    jobclass{k} = ClosedClass(model, ['Class',int2str(k)], N(k), queue);
    queue.setService(jobclass{k}, Erlang.fitMeanAndOrder(k,2));
end

P = model.initRoutingMatrix();
P{jobclass{1},jobclass{2}}(queue,queue) = 1.0;
P{jobclass{2},jobclass{3}}(queue,queue) = 1.0;
P{jobclass{3},jobclass{1}}(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

```
>> ctmcAvgTable = SolverCTMC(model).getAvgTable
ctmcAvgTable =
  3x6 table
    Station     Class       QLen       Util      RespT      Tput
    _______    ________    _______    _______    _____    _______
    'Queue'    'Class1'    0.16667    0.16667      1      0.16667
    'Queue'    'Class2'    0.33333    0.33333      2      0.16667
```

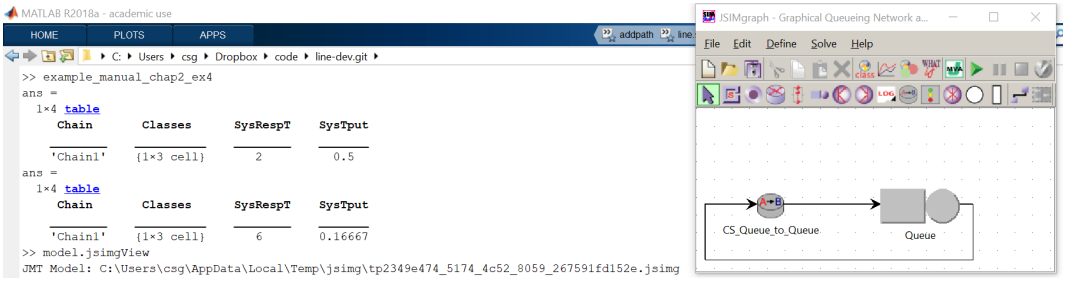


Figure 2.4: Re-entrant lines as an example of class-switching

```
    'Queue'    'Class3'        0.5          0.5        3        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 `getAvgSysTable` method, i.e.,

```
>> ctmcAvgSysTable = SolverCTMC(model).getAvgSysTable
ctmcAvgSysTable =
  1x4 table
     Chain        Classes       SysRespT     SysTput
  _________    ___________    _________    _______
  'Chain1'     {1x3 cell}        2           0.5
```

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

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

```
jobclass{1}.completes = false;
jobclass{2}.completes = false;
```

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

```
>> ctmcAvgSysTable2 = SolverCTMC(model).getAvgSysTable
ctmcAvgSysTable =
  1x4 table
     Chain        Classes       SysRespT     SysTput
  _________    ___________    _________    _______
  'Chain1'     {1x3 cell}        6          0.16667
```

### 2.2.7 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:

```
model = Network('QNC');
% Block 1: nodes
clientDelay = Delay(model, 'Client');
cacheNode = Cache(model, 'Cache', 1000, 50, ReplacementStrategy.LRU);
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:

```
% Block 2: classes
clientClass = ClosedClass(model, 'ClientClass', 1, clientDelay, 0);
hitClass = ClosedClass(model, 'HitClass', 0, clientDelay, 0);
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));
```

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`

```
% Block 3: topology
P = model.initRoutingMatrix();
P{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{hitClass, hitClass}(cacheNode, cacheDelay) = 1.0;
P{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{hitClass, clientClass}(cacheDelay, clientDelay) = 1.0;
P{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 MATLAB-based simulation engine provided within LINE, the `SSA` solver:

```
% Block 4: solution
ssaAvgTable = SolverSSA(model,'samples',2e4,'seed',1,'verbose',true).getAvgTable
```

The above script produces the following result

```
SSA samples:   20000
SSA analysis completed in 11.902675 sec
ssaAvgTable =
 3x6 table
     Station           Class          QLen       Util      RespT      Tput
    _________    _______________    _______    _______    _____    _______
    'Client'     'ClientClass'            0          0        0     2.9792
    'CacheDelay' 'HitClass'         0.50045    0.50045      0.2     2.5022
    'CacheDelay' 'MissClass'        0.49955    0.49955        1    0.49955
```

The departing flows from the `CacheDelay` are the miss and hit rates. Thus, the hit rate is 2.4554 req/ms, while the miss rate is 0.50892 req/ms, which both include the service times.

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` based on MATLAB's `spmd` construct:

```
ssaAvgTablePara = SolverSSA(model,'para','samples',2e4,'seed',1).getAvgTable
```

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

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 2 workers.
ssaAvgTablePara =
 3x6 table
     Station           Class          QLen       Util      RespT      Tput
    _________    _______________    _______    _______    _____    _______
    'Client'     'ClientClass'            0          0        0     2.8629
    'CacheDelay' 'HitClass'         0.47533    0.47533      0.2     2.3767
    'CacheDelay' 'MissClass'        0.52467    0.52467        1    0.52467
```

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.

### 2.2.8 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.

```
model = Network('model');
% Block 1: nodes
node{1} = Delay(model, 'Delay');
node{2} = 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.

```
% Block 2: classes
jobclass{1} = ClosedClass(model, 'Class1', 5, node{1}, 0);
node{1}.setService(jobclass{1}, Exp(1.0));
node{2}.setService(jobclass{1}, Exp(0.5));

% Block 3: topology
model.link(Network.serialRouting(node{1},node{2}));
```

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

```
% Block 4: solution
RDfluid = SolverFluid(model).getCdfRespT();
RDsim = SolverJMT(model,'seed',23000,'samples',1e4).getCdfRespT();
```

The returned data structures, `RDfluid` and `RDsim`, are cell arrays consisting of a 2-column matrix. The element in position $\{i, r\}$ of the cell array describes the response times at station $i$ for class $r$. The two columns within such matrices are defined similar to the output of `MATLAB`'s `ecdf` function, i.e., 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.

For example, to plot the complementary CDF $1 - F(t)$ we can use the following code

```
% Plot results
semilogx(RDsim{2,1}(:,2),1-RDsim{2,1}(:,1),'r'); hold all;
semilogx(RDfluid{2,1}(:,2),1-RDfluid{2,1}(:,1),'--');
legend('jmt-transient','fluid-steady','Location','Best');
ylabel('Pr(T > t)'); xlabel('time t');
```

which produces the graph shown in Figure 2.5 The graph shows that, although the simulation refers to a

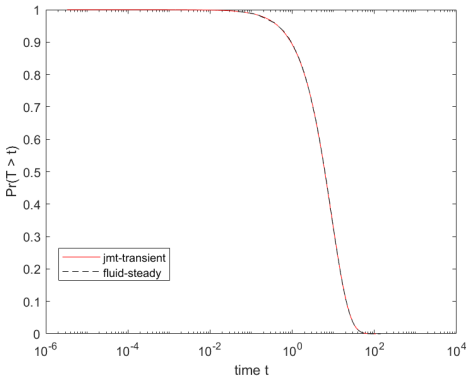


Figure 2.5: Comparison of simulated response time distribution and its fluid approximation

transient, while the fluid approximation refers to steady-state, there is a tight matching between the two response time distributions.

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

```
>> prc95=max(RDsim{2,1}(RDsim{2,1}(:,1)<0.95,2))
```

```
prc95 =
   27.0222
>> prc99=max(RDsim{2,1}(RDsim{2,1}(:,1)<0.99,2))
prc99 =
   41.8743
```

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.

### 2.2.9 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.

We first create a MATLAB function (e.g., `ex8.m`) with header

```
function p_star = ex8()
```

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

```
model = Network('LoadBalCQN');
% Block 1: nodes
delay = Delay(model,'Think');
queue1 = Queue(model, 'Queue1', SchedStrategy.PS);
queue2 = Queue(model, 'Queue2', SchedStrategy.PS);
```

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

```
% Block 2: classes
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:

```
% Block 3: topology
P = model.initRoutingMatrix();
P{cclass}(queue1, delay) = 1.0;
P{cclass}(queue2, delay) = 1.0;
model.link(P);
```

We now write a nested function that returns the system response time for the jobs as a function of the routing probability $p$ to choose queue 1 instead of queue 2:

```
% Block 4: solution
    function R = objFun(p)
        P{cclass}(delay, queue1) = p;
        P{cclass}(delay, queue2) = 1-p;
        model.link(P);
        R = SolverMVA(model,'exact','verbose',false).getAvgSysRespT;
    end
p_opt = fminbnd(@(p) objFun(p), 0,1)
```

In the above listing, `objFun` first updates the routing topology, prior to obtaining the corresponding system response time. To search for the optimal routing probability $p$, we have also used MATLAB's `fminbnd` which restricts the search range in the interval $[0, 1]$.

We are now ready to run `ex8` from the MATLAB command line. The execution returns the optimal value `p_opt`$= 0.6105$.

### 2.2.10 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.

```
[model,source,queue,sink,oclass]=gallery_merl1;
```

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

```
solver = SolverCTMC(model,'cutoff',100,'seed',23000);
sa = solver.sampleSysAggr(5e4);
```

The returned data structure supplies information about the stateful nodes (here `source` and `queue`) at each of the 50,000 instants of sampling, together with the events that have been collected at these instants.

```
>> sa
sa =
  struct with fields:

          handle: {[1x1 Source]  [1x1 Queue]}
               t: [50000x1 double]
           state: {[Inf]  [50000x1 double]}
           event: {1x100000 cell}
    isaggregate: 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: 2`) followed by an arrival event at node 2 (the type `EventType.ARV` maps to `event: 1`) which accepts it always (`prob: 1`).

```
>> sa.event{1}
ans =
  Event with properties:

     node: 1
    event: 2
    class: 1
     prob: NaN
    state: []
        t: 0
>> sa.event{2}
ans =
  Event with properties:

     node: 2
    event: 1
    class: 1
     prob: 1
    state: []
        t: 0
```

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

```
ind = model.getNodeIndex(queue);
filtEvent = cellfun(@(c) c.node == ind && c.event == EventType.DEP, sa.event);
```

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

```
interDepTimes = diff(cellfun(@(c) c.t, {sa.event{filtEvent}}));
```

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

```
SCVdEst = var(interDepTimes)/mean(interDepTimes)^2
```

which evaluates to $0.8787$. Using Marshall's exact formula for the $GI/G/1$ queue we get a theoretical value of $0.8750$, the calculation is included in the script associated to this example (`getting_started_ex9.m`).

# Chapter 3

# Network models

Throughout this chapter, we discuss the specification of `Network` models, which are extended queueing networks. LINE currently support 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 mean number of jobs in execution at the station.

- Mean response time (`RespT`). This is the mean time a job spends to traverse 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 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.

The above metrics refer to 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:

```
model = 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 list of nodes available in `Network` models is given in Table 3.1.1.

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 |

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.

```
queue = Queue(model, 'Queue1', SchedStrategy.FCFS);
```

specifies a first-come first-served 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, however 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-served (`SchedStrategy.FCFS`)

- Infinite-server (`SchedStrategy.INF`)
- Processor-sharing (`SchedStrategy.PS`)
- 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 (`SchedStrategy.HOL`)

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

```
delay = Delay(model, 'ThinkTime');
```

As for queues, for readability it is possible to instantiate delay nodes using the `DelayStation` constructor, which is entirely equivalent to the one of 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:

```
source = Source(model, 'Source');
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

```
fork = Fork(model, 'Fork');
join = Join(model, 'Join');
```

The number of tasks sent by a `Fork` on each output link can be set using the `setTasksPerLink` method of the `fork` object.

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 need to be created in order to send that class of jobs only to $A$ and $B$.

**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, or that a class 2 job should become of class 1 with probability 0.3. This example is instantiated as follows

```
cs = ClassSwitch(model, 'ClassSwitchPoint',[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.,

```
cacheNode = Cache(model, 'Cache1', nitems, capacity, ReplacementStrategy.LRU);
```

If the capacity is an 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 (e.g., `[10,5]`) then the node is a list-based cache, where the vector entries specify the capacity of each list. We point to [16] 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.RAND`)
- 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 receive service; it is instead directly forwarded to the next node according to the specified routing strategy. A `Router` can be instantiated as follows:

```
router = Router(model, 'RouterNode');
```

An example of use of this node is given in Section 2.2.5. 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.RR`)
- 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 timestamp of passage and general information about the jobs. The node can be instantiated as follows

```
logger=Logger(self,'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 identificator 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 last passage of a job of the same class.
- `setTimeAnyClass`: record the time elapsed since 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), 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,

```
example_closedModel_3
delay.setChainCapacity([1,1])
model.refreshCapacity()
```

creates an example model with two chains and three classes (specified in `example_closedModel_3.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 `setChainCapacity([1,7])`, 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 `example_closedModel_3` 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 need for a `refreshCapacity` call, since the internal representation of the `Network` object used by the solvers is still to be created.

### 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`

```
source = Source(model, 'Source');
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

```
class1 = OpenClass(model, 'Class1');
```

LINE does not require to explicitly associate source and sink with the open classes in their constructors, as this is done automatically. However, the LINE language requires to explicitly create 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 includes 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.:

```
class2 = 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 system performance index (system response time). The reference station of an open class is always automatically set by LINE to be the `Source`. 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.

```
model = Network('model');
% Block 1: nodes
delay = Delay(model, 'Delay');
queue = Queue(model, 'Queue1', SchedStrategy.FCFS);
% Block 2: classes
cclass = ClosedClass(model, 'Class1', 10, delay, 0);
slclass = 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
P = model.initRoutingMatrix;
P{cclass} = [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 both 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.,

```
class2 = 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 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 to choose 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 throuhput corresponds to the rate at which jobs arrive to 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 contain a single class, thus per-chain and per-class performance indexes will be identical.

### 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 to specify 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 switching of class for jobs moving between stations (`ClassSwitch` elements).

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

```
P = model.initRoutingMatrix;
P{class1}(source,queue) = 1.0;
P{class1}(queue,[queue,delay]) = [0.3,0.7]; % self-loop with probability 0.3
P{class1}(delay,sink) = 1.0;
P{class2}(delay,queue) = 1.0; % note: closed class jobs start at delay
P{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{class1} = [0,1,0,0;   % row: source
             0,.3,.7,0; % row: queue
             0,0,0,1;   % row: delay
             0,0,0,0];  % row: sink
P{class2} = [0,0,0,0;
```

```
                0,0,1,0;
                0,1,0,0;
                0,0,0,0];
model.link(P);
```

Where needed, the `getClassIndex` and `getNodeIndex` functions return the numerical index associated to a node name, for example `model.getNodeIndex('Delay')`. 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 `getClassNames`, `getStationNames`, and `getNodeNames` functions of the `Network` class.

It is also important to note that the routing matrix in the last example is specified between *nodes*, instead than between just stations or stateful nodes, which means that for example elements such as the `Sink` need to be explicitly considered in the routing matrix. The only exception is that `ClassSwitch` elements do not need to be explicitly instantiated and explicited 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 screen in human-readable format using the `printRoutingMatrix` function, e.g.,

```
>> model.printRoutingMatrix
Delay [Class1] => Queue1 [Class1] : Pr=1.000000
Delay [Class2] => Queue1 [Class2] : Pr=0.001000
Queue1 [Class1] => Queue1 [Class1] : Pr=0.300000
Queue1 [Class1] => Source [Class1] : Pr=0.700000
Queue1 [Class2] => Source [Class2] : Pr=1.000000
Source [Class1] => Sink [Class1] : Pr=1.000000
Source [Class2] => Queue1 [Class2] : Pr=1.000000
Sink [Class2] => Source [Class2] : Pr=1.000000
```

### 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.RR`). 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.RR);
```

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 startegies. 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

```
lambda = [10,20]; % lambda(r) - arrival rate of class r
D = [11,12; 21,22]; % D(i,r) - class-r demand at station i (PS)
Z = [91,92; 93,94]; % Z(i,r)  - class-r demand at station i (INF)
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 to replace the arrival rate vector `A` by a vector `N` specifying the job populations for each of the closed classes, e.g.,

```
N = [10,20]; % N(r) - closed population in class r
D = [11,12; 21,22]; % D(i,r) - class-r demand at station i (PS)
modelPsInf = Network.cyclicPs(N,D)
```

### 3.1.5 Class switching

Depending on the specified probabilities, a job will be able to switch 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 next.

#### 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:

```
class3 = ClosedClass(model, 'Class3', 0, queue, 0);

P = model.initRoutingMatrix;
P{class1,class1}(source, queue) = 1.0;
P{class1,class1}(queue, [queue,delay]) = [0.3,0.7];
P{class1,class1}(delay, sink) = 1.0;
P{class2,class3}(delay, queue) = 1.0;
```

```
P{class3,class2}(queue, delay) = 1.0;
model.link(P);
```

where `P{r,s}` is the routing matrix for jobs switching from class `r` to `s`. That is, `P{r,s}(i,j)` is the probability that a job in class `r` departs node `i` routing into node `j` as a job of class `s`.

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, one needs 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 this node requires to specify a probability matrix $C$ such that $C(r, s)$ is the probability that a job of class $r$ arriving into the `ClassSwitch` 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

```
C = [0, 1; 0, 1];
node = ClassSwitch(model, 'CSNode',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 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 wether 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,

```
refModel = 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 `Source` station. The class `PhaseType` 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 case the following distributions:

- Exponential distribution: `Exp`$(\lambda)$, where $\lambda$ is the rate of the exponential
- $n$-phase Erlang distribution: `Erlang`$(\alpha, n)$, where $\alpha$ 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 $\lambda_1$ with probability $p$, and an exponential with rate $\lambda_2$ otherwise.
- $n$-phase hyper-exponential distribution: `HyperExp`$(p, \lambda)$, 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`$(\mu_1, \mu_2, \phi_1)$, which assigns phases $\mu_1$ and $\mu_2$ to the two rates, and completion probability from phase 1 equal to $\phi_1$ (the probability from phase 2 is $\phi_2 = 1.0$).
- $n$-phase Coxian distribution: `Coxian`$(\mu, \phi)$, 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`$(\alpha, T)$, 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/$\mu^2$, 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));
```

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 `PhaseType.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`$(\mu)$ assigns probability 1.0 to the value $\mu$.

- Uniform distribution: `Uniform`$(a, b)$ assigns uniform probability $1/(b-a)$ to the interval $[a, b]$.
- Gamma distribution: `Gamma`$(\alpha, k)$ assigns a gamma density with shape $\alpha$ and scale $k$.
- Pareto distribution: `Pareto`$(\alpha, k)$ assigns a Pareto density with shape $\alpha$ 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 `NaN`.

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

### Fitting a distribution

The `fitMeanAndSCV` function is available for all distributions that inherit from the `PhaseType` 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`$(\mu, n)$, which instantiates a $n$-phase Erlang with given mean $\mu$.

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 an 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.,

```
>> dist = Exp(1);
>> dist.getMean
ans =
     1
>> dist.getSCV
ans =
     1
```

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

```
>> dist.sample(3)
ans =
    0.2049
    0.0989
    2.0637
```

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

```
>> dist.evalCDFInterval(2,5)
ans =
    0.1286
>> dist.evalCDF(5)-dist.evalCDF(2)
ans =
    0.1286
```

For more advanced uses, the distributions of the `PhaseType` 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]. The result will be a cell array where element $k+1$ corresponds to matrix $D_k$.

#### 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 inherit from the `PointProcess` class, and 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 `PhaseType` class.

### 3.1.7 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 Figure 3.1 below. 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 MATLAB graph object, e.g.

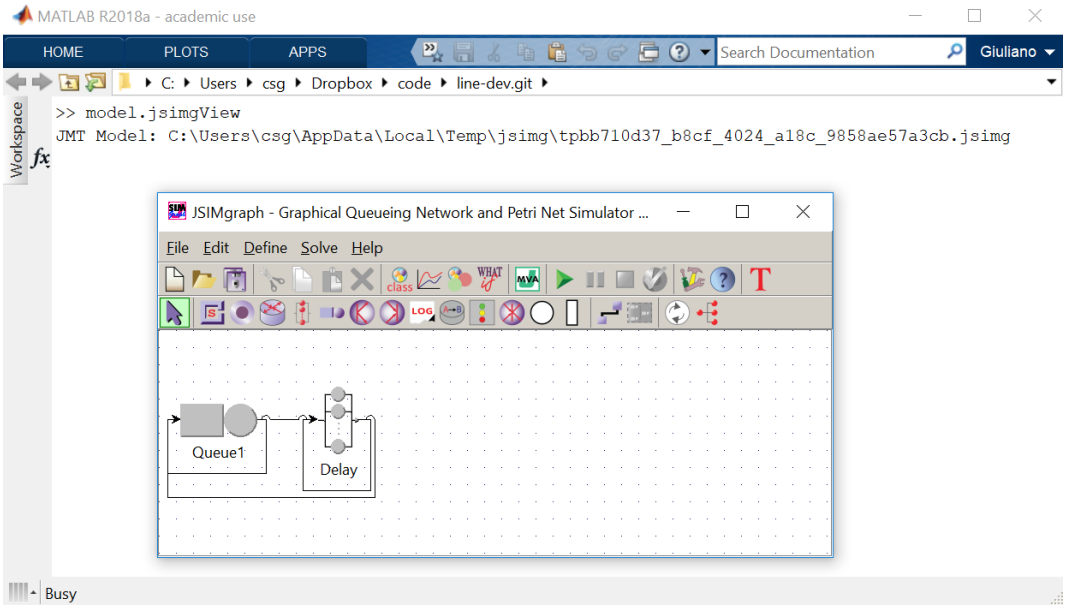


Figure 3.1: jsimgView function

```
G = model.getGraph();
plot(G,'EdgeLabel',G.Edges.Weight,'Layout','Layered')
```

plots a graph of the network topology in term of stations only. In a similar manner, the following variant of the same command shows the model in terms of nodes, which corresponds to the internal representation within LINE.

```
[~,H] = model.getGraph();
plot(H,'EdgeLabel',H.Edges.Weight,'Layout','Layered')
```

Figure 3.2 shows the difference between the two commands for an open queueing network with two classes and class-switching. Weights on the edges correspond to routing probabilities. In the station topology on the left, note that since the `Sink` node is not a station, departures to the `Sink` are drawn as returns to the `Source`. The node topology on the right, illustrates all nodes, including `ClassSwitch` nodes that are automatically added by LINE to apply the class-switching routing strategy. Double arcs between nodes indicate that both classes are routed to the destination.

Furthermore, the graph properties concisely summarize the key features of the network

```
>> G.Nodes
ans =
  2x5 table
```

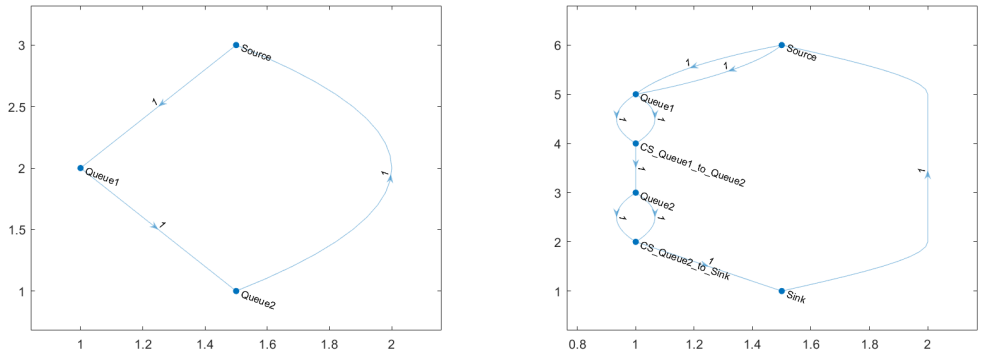


Figure 3.2: `getGraph` function: station topology (left) and node topology (right) for a 2-class tandem queueing network with class-switching.

```
    Name          Type           Sched     Jobs     ClosedClass1
    _______    ____________    _______    ____    ____________

    'Delay'    'Delay'         'inf'      5           1
    'Queue1'   'Queue'         'ps'       0           2
>> G.Edges
ans =
  3x4 table
          EndNodes           Weight     Rate        Class
    ____________________     ______    _____    _____________

    'Delay'    'Delay'        0.7         1      'ClosedClass1'
    'Delay'    'Queue1'       0.3         1      'ClosedClass1'
    'Queue1'   'Delay'          1       0.5      'ClosedClass1'
```

Here, `Edge.Weight` is the routing probability between the nodes, whereas `Edge.Rate` is the service rate of the node appearing in the first column under `EndNodes`.

## 3.2 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.
- `PMIF2LINE` imports a XML file containing a PMIF 1.0 model.

Both scripts require in input the filename and desired model name, and return a single output, e.g.,

```
qn = PMIF2LINE([pwd,'\\examples\\data\\PMIF\\pmif_example_closed.xml'],'Mod1')
```

where qn is an instance of the Network class.

Network object can be saved in binary .mat files using MATLAB's standard save command. However, it is also possible to export a textual script that will dynamically recreate the same Network object. For example,

```
example_closedModel_1; LINE2SCRIPT(model, 'script.m')
```

creates a new file script.m with code

```
model = Network('model');

%% Block 1: nodes
node{1} = DelayStation(model, 'Delay');
node{2} = Queue(model, 'Queue1', SchedStrategy.FCFS);

%% Block 2: classes
jobclass{1} = ClosedClass(model, 'Class1', 10, node{1}, 0);

node{1}.setService(jobclass{1}, Exp.fitMean(1.000000)); % (Delay,Class1)
node{2}.setService(jobclass{1}, Exp.fitMean(1.500000)); % (Queue1,Class1)

%% Block 3: topology
P = model.initRoutingMatrix(); % initialize routing matrix
P{1,1}(1,1) = 7.000000e-01; % (Delay,Class1) -> (Delay,Class1)
P{1,1}(1,2) = 3.000000e-01; % (Delay,Class1) -> (Queue1,Class1)
P{1,1}(2,1) = 1; % (Queue1,Class1) -> (Delay,Class1)
model.link(P);
```

that is equivalent to the model specified in example_closedModel_1.m.

### 3.2.1 Creating a LINE model using JMT

Using the features presented in the previous section, one can create a model in JMT and automatically derive a corresponding LINE script from it. For instance, the following command performs the import and translation into a script, e.g.,

```
LINE2SCRIPT(JMT2LINE('myModel.jsimg'),'myModel.m')
```

transforms and save the given JSIMgraph model into a corresponding LINE model.

LINE also provides two static functions to inspect jsimg and jsimw files before conversion, called SolverJMT.jsimgOpen and SolverJMT.jsimwOpen require as an input parameter only the JMT file name, e.g., 'myModel.jsimg'.

It is also possible to automate the editing and import of JMT models from MATLAB using the jsimgEdit command. This will open an empty JMT model and upon save this will be automatically reimported into MATLAB.

Table 3.2: Supported JSIM features for automated model import and analysis

| JMT Feature | Support | Notes |
|---|---|---|
| Distributions | Full | Phase-Type, Deterministic, Exponential, Erlang, Gamma, Hyperexponential, Coxian, Pareto, Uniform, Zero Service Time, Replayer |
| Classes | Full | Open class, Closed class, Class priorities |
| Metrics | Full | Number of customers, Residence Time, System Throughput, System Response Time, Throughput, Throughput per sink, Utilization |
| Nodes | Full | ClassSwitch, Delay, Logger, Queue, Router |
| Routing | Full | Random, Probabilities, Round Robin, Join the Shortest Queue |
| Scheduling | Full | FCFS, HOL, LCFS, RAND, SJF, SEPT, LJF, LEPT, PS, DPS, GPS |
| Nodes | Partial | Fork, Join, Source, Sink |
| Distributions | No | Burst (General), Burst (MAP), Burst (MMPP2), Normal |
| Nodes | No | Finite Capacity Region, Place, Scaler, Semaphore, Transition |
| Routing | No | Shortest Response Time, Least Utilization, Fastest Service, Load Dependent |
| Metrics | No | Drop rate, Response time per sink, power |
| Scheduling | No | Load Dependent |

### 3.2.2 Supported JMT features

Table 3.2 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:

- LINE does not support general phase-type distributions, rather the most general supported class are acyclic phase-type (APH) distributions with an arbitrary number of phases.
- `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 sinlge `Source` can be instantiated in a LINE model, whereas there is no such constraint in JMT.

# Chapter 4

# Analysis methods

## 4.1 Steady-state analysis

### 4.1.1 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,

```
solver = 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
QN = solver.getAvgQLen()
% UN(i,r): utilization of class r at station i
UN = solver.getAvgUtil()
% RN(i,r): mean response time of class r at station i (summed on visits)
RN = solver.getAvgRespT()
% TN(i,r): mean throughput of class r at station i
TN = solver.getAvgTput()
```

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

```
[QN,UN,RN,TN] = solver.getAvg()
```

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

```
AvgTable = solver.getAvgTable()
```

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 `getAvg` 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(1,2)
QN(queue,cclass)
QN(model.getStationIndex('MyQueue'),model.getClassIndex('MyClass'))
```

Similar methods are defined to obtain aggregate performance metrics at chain level at each station, namely `getAvgQLenChain` for queue-lengths, `getAvgUtilChain` for utilizations, `getAvgRespTChain` for response times, `getAvgTputChain` for throughputs, and the `getAvgChain` method to obtain all the previous metrics.

### 4.1.2 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. For example, the following code plots the cumulative distribution function at steady-state for class 1 jobs when they visit station 2:

```
solver = SolverFluid(model);
FC = solver.getCdfRespT();
plot(FC{2,1}(:,2),FC{2,1}(:,1)); xlabel('t'); ylabel('Pr(RespT<t)');
```

### 4.1.3 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 `getAvgSys` and `getAvgSysTable` 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 the user 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.

```
class1 = ClosedClass(model, 'ClosedClass1', 1, queue, 0);
class2 = ClosedClass(model, 'ClosedClass2', 0, queue, 0);

class1.completes = false;

P = cell(2); % 2-classes model
P{1,1} = [0,1; 0,0]; % routing within class 1 (no switching)
P{1,2} = [0,0; 1,0]; % routing from class 1 into class 2
P{2,1} = [0,0; 1,0]; % routing within class 2 (no switching)
P{2,2} = [0,1; 0,0]; % routing from class 2 into class 2

model.link(P);
```

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

## 4.2 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.

### 4.2.1 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 `PhaseType`. 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 $c_1 = 0$.

- $n_r$: 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 `PhaseType`. 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. All state variables are non-negative integers. The `SchedStrategy.EXT` policy is 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 $r = 1, ..., R$.

Table 4.1: State descriptors for Markovian scheduling policies

| **Sched. strategy** | **Station state vector** | **State condition** |
|---|---|---|
| `EXT` | $[\texttt{Inf}, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$ | $\sum_k s_{rk} = 1, \forall r$ |
| `FCFS`, `HOL`, `LCFS` | $[c_b, ..., c_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$ | $\sum_r \sum_k s_{rk} = 1$ |
| `SEPT`, `RAND` | $[b_1, ..., b_R, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$ | $\sum_r \sum_k s_{rk} = 1$ |
| `PS`, `DPS`, `GPS`, `INF` | $[s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$ | None |

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 `example_closedModel_4.m`. 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:

```
>> example_closedModel_4;
>> space = State.fromMarginalAndRunning(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4     3     2     1     0     0     0
     4     3     2     0     1     0     0
     3     4     2     1     0     0     0
     3     4     2     0     1     0     0
```

Here, each row of `space` corresponds to a valid state. The argument `[2,1,1,1]` gives the number of jobs in the node for the 4 classes, while `[2,1,0,0]` 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

```
>> space = State.fromMarginalAndStarted(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4     3     2     1     0     0     0
     3     4     2     1     0     0     0
```

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.

```
>> space = State.fromMarginal(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4     3     2     1     0     0     0
     4     3     2     0     1     0     0
     4     2     2     0     0     1     0
     4     1     1     1     0     1     0
     4     1     1     0     1     1     0
     3     4     2     1     0     0     0
     3     4     2     0     1     0     0
     3     2     2     0     0     0     1
     3     1     1     1     0     0     1
     3     1     1     0     1     0     1
     2     4     2     0     0     1     0
     2     3     2     0     0     0     1
     2     1     1     0     0     1     1
     1     4     1     1     0     1     0
```

```
    1     4     1     0     1     1     0
    1     3     1     1     0     0     1
    1     3     1     0     1     0     1
    1     2     1     0     0     1     1
    1     1     0     1     0     1     1
    1     1     0     0     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` of can be used to check and change the prior probabilities for the initial states specified for a station or stateful node.

### 4.2.2 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 with elements $\mathtt{n}(i, r)$ specifying the total number of resident class-$r$ jobs at node $i$ and the latter two require a matrix $\mathtt{s}(i, r)$ 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.

### 4.2.3 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 position time $t' > t$ without ever being visited again by any job. `LINE` allows one 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 an 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 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 $E[n(t)|s_0]$.

### 4.3.1 Computing transient averages

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

```
[Qt,Ut,Tt] = model.getTranHandles();
```

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

```
[QNt,UNt,TNt] = SolverCTMC(model,'timespan',[0,1]).getTranAvg(Qt,Ut,Tt);
plot(QNt{1,1}.t, QNt{1,1}.metric)
```

The transient average queue-length at node $i$ for class $r$ is stored within `QNt{i,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

```
>> SolverCTMC(model).getAvgTable
State space size: 11 states.
CTMC analysis (method: default) completed in 0.017275 seconds.
ans =
  2x6 table
    Station      Class      QLen      Util      RespT      Tput
    _________    _________  ______    _______   _______    _______
    'Delay'      'Class1'   2.222       2.222         1      2.222
    'Queue1'     'Class1'   7.778     0.99991    11.668    0.66661
>> SolverCTMC(model,'timespan',[0,1]).getAvgTable
State space size: 11 states.
CTMC analysis completed in 0.017475 sec
ans =
  2x6 table
    Station      Class       QLen      Util      RespT     Tput
    _________    _________   ______    _______   _____     _______
    'Delay'      'Class1'    7.7864     7.7864       0      7.7864
    'Queue1'     'Class1'    2.2136    0.89021       0     0.59348
```

### 4.3.2 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`.

## 4.4 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 aggragted 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:

```
>> model=gallery_aphm1; samplePath = ...
    SolverJMT(model).sampleAggr(model.nodes{2},1e1); [samplePath.t, ...
    samplePath.state]
JMT Model: /tmp/jsimg/tpa52644b6_17ce_46ee_9c09_8da7c2d1d3c6.jsimg
ans =
         0         0
    0.3255    1.0000
    0.3562         0
    1.1478    1.0000
    1.2220         0
    1.8727    1.0000
    2.6076    2.0000
    2.6862    3.0000
    3.4737    2.0000
    3.7663    3.0000
    4.2110    2.0000
```

In the example, `samplePath.t` 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:

```
>> model=gallery_aphm1; samplePath = SolverSSA(model).sampleSys(10); ...
    [samplePath.t, samplePath.state{1}, samplePath.state{2}]

SSA analysis (method: default) completed in 0.011636 seconds.
ans =
    0.4776       Inf    1.0000         0         0         0
```

```
    0.5843       Inf         0    1.0000         0         0
    1.2043       Inf    1.0000         0         0    1.0000
    1.2952       Inf    1.0000         0         0         0
    1.3196       Inf         0    1.0000         0         0
    1.5724       Inf    1.0000         0         0    1.0000
    1.5975       Inf         0    1.0000         0    1.0000
    1.6129       Inf         0    1.0000         0         0
    1.6226       Inf    1.0000         0         0    1.0000
    2.5289       Inf    1.0000         0         0         0
```

## 4.5 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 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 several models.

### 4.5.1 Internal representation of the model structure

For efficiency reasons, once a user requests to solve a `Network`, LINE calls internally generates 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 multiclass 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

```
qn = model.getStruct()
```

The table below presents the properties of the `NetworkStruct` class.

### 4.5.2 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

Table 4.2: `NetworkStruct` properties

| Field | Type | Description |
|---|---|---|
| `cap`$(i)$ | `integer` | Total capacity at station $i$ |
| `chains`$(c,r)$ | `logical` | `true` if class $r$ is in chain $c$, or `false` otherwise |
| `classcap`$(i,r)$ | `integer` | Maximum buffer capacity available to class $r$ at station $i$ |
| `classname`$\{r\}$ | `string` | Name of class $r$ |
| `classprio`$(r)$ | `integer` | Priority of class $r$ (0 = highest priority) |
| `csmask`$(r,s)$ | `logical` | true if class $r$ can switch into class $s$ at some node |
| `isstation`$(i)$ | `logical` | true if node $i$ is a station |
| `isstateful`$(i)$ | `logical` | true if node $i$ is a stateful node |
| `lst`$\{i,r\}$ | `function handle` | Laplace-Stieltjes transform of the service or arrival distribution for class $r$ at station $i$ |
| `mu`$\{i,r\}(k)$ | `double` | Service or arrival rate in phase $k$ for class $r$ at station $i$, with `mu`$\{i,r\}$ =`NaN` if `Disabled` and `mu`$\{i,r\} = 10^7$ if `Immediate`. |
| `nchains` | `integer` | Number of chains in the network |
| `nclasses` | `integer` | Number of classes in the network |
| `nclosedjobs` | `integer` | Total number of jobs in closed classes |
| `njobs`$(r)$ | `integer` | Number of jobs in class $r$ (`Inf` for open classes) |
| `nnodes` | `integer` | Number of nodes in the network |
| `nservers`$(i)$ | `integer` | Number of servers at station $i$ |
| `nstations` | `integer` | Number of stations in the network |
| `nstateful` | `integer` | Number of stateful nodes in the network |
| `nodenames`$\{i\}$ | `string` | Name of node $i$ |
| `nodetypes`$\{i\}$ | `string` | Type of node $i$ (e.g., `NodeType.Sink`) |
| `nvars` | `integer` | Number of local state variables at stateful nodes |
| `phases`$(i,r)$ | `integer` | Number of phases for service process of class $r$ at station $i$ |
| `phasessz`$(i,r)$ | `integer` | Number of state vector elements used to describe phase |
| `phaseshift`$(i,r)$ | `integer` | Position shift to read phase element in state |
| `proc`$\{i,r\}$ | `cell` | Matrix representation of [1] the class $r$ service process at station $i$ |
| `phi`$\{i,r\}(k)$ | `double` | Completion probability in phase $k$ for class $r$ at station $i$ |
| `pie`$\{i,r\}(k)$ | `double` | Entry probability in phase $k$ for class $r$ at station $i$ |
| `rates`$(i,r)$ | `double` | Service rate of class $r$ at station $i$ (or arrival rate if $i$ is a `Source`) |
| `refstat`$(r)$ | `integer` | Index of reference station for class $r$ |
| `rt`$(idx_{ir}, idx_{js})$ | `double` | Probability of routing from stateful node $i$ to $j$, switching class from $r$ to $s$ where, e.g., $idx_{ir} = (i-1) * \texttt{nclasses} + r$. |
| `rtnodes`$(idx_{ir}, idx_{js})$ | `double` | Same as `rt`, but $i$ and $j$ are nodes, not necessarily stateful ones. |
| `rtfun`$(\texttt{st1}, \texttt{st2})$ | `matrix` | State-dependent routing table given initial (`st1`) and final (`st2`) state cell arrays. Table entries defined as in `rt`. |
| `schedparam`$(i,r)$ | `double` | Parameter for class $r$ strategy at station $i$ |
| `sched`$\{i\}$ | `cell` | Scheduling strategy at station $i$ (e.g., `SchedStrategy.PS`) |
| `schedid`$(i)$ | `integer` | Scheduling strategy id at station $i$ (e.g., `SchedStrategy.ID_PS`) |
| `sync`$\{s\}$ | `struct` | Data structure specifying a synchronization $s$ among nodes |
| `scv`$(i,r)$ | `double` | Squared coefficient of variation of class $r$ service times at station $i$ (or inter-arrival times if station $i$ is a `Source`) |
| `space`$\{t\}$ | `integer` | 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`$\{i\}$ | `integer` | Current state of stateful node $i$. This field may be initially empty and updated by the solver during execution. |
| `visits`$\{c\}(i,r)$ | `double` | Number of visits that a job in chain $c$ pays to node $i$ in class $r$ |
| `varsparam`$\{i\}$ | `double` | Parameters for local variable instantiation at stateful node $i$ |

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`, `schedid`, and `schedparam` fields.
- `refreshService`: 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, Exp(1.0));
model.refreshService;
```

### 4.5.3 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

```
>> model = Network('model');
node{1} = ClassSwitch(model,'CSNode',[0,1;0,1]);
node{2} = Queue(model, 'Queue1', SchedStrategy.FCFS);
>> model.getNodes
ans =
  2x1 cell array
    {1x1 ClassSwitch}
    {1x1 Queue}
>> model.resetNetwork
ans =
  1x1 cell array
    {1x1 Queue}
```

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.

### 4.5.4 Saving a network object before a change

The `Network` 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

```
modelByRef = model; modelByRef.setName('myModel1');
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 `Network` 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:

```
SolverMVA(model,'exact').getAvgTable()
SolverMVA(model,'method','exact').getAvgTable()
opt = SolverMVA.defaultOptions; opt.method = 'exact'; ...
    SolverMVA(model,opt).getAvgTable()
```

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 `Network` solvers are available within LINE 2.0.0:

- `AUTO`: 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 `SolverAuto` 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 [22]. 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 [2]. 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 discrete-event simulator 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 `Network` solvers. Table 5.2 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.:

```
>> solver = Solver.load('mg1',model);
>> solver.getAvgTable();
```

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

```
>> solver = Solver.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:

```
>> solver = Solver.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] |
| CTMC | gpu | Solution based on global balance run on GPU if available | – |
| FLUID | default | ODE-based mean field approximation | [23] |
| JMT | default | Alias for the jsim method | – |
| JMT | jmva | Alias for the jmva.mva method | – |
| JMT | jmva.mva | Exact MVA in JMVA | [24] |
| JMT | jmva.recal | Exact RECAL algorithm in JMVA | [13] |
| JMT | jmva.comom | Exact CoMoM algorithm in JMVA | [7] |
| JMT | jmva.amva | Approximate MVA, alias for jmva.bs. | – |
| JMT | jmva.aql | AQL algorithm in JMVA | [29] |
| JMT | jmva.bs | Bard-Schweitzer algorithm in JMVA | [5, §9.1.1] |
| JMT | jmva.chow | Chow algorithm in JMVA | [12] |
| JMT | jmva.dmlin | De Souza-Muntz Linearizer in JMVA | [14] |
| JMT | jmva.lin | Linearizer algorithm in JMVA | [11] |
| JMT | jmva.ls | Logistic sampling in JMVA | [8] |
| JMT | jsim | Exact discrete-event simulation in JSIM | [2] |
| MAM | default | Matrix-analytic solution of structured QBDs | [18] |
| MAM | dec.source | Decomposition based on source arrival flow | – |
| MAM | dec.poisson | Decomposition based on Poisson arrival flow | – |
| MVA | default | Approximation, method depends on model | – |
| MVA | amva | Bard-Schweitzer approximate MVA | [5, §9.1.1] |
| MVA | exact | Exact solution, method depends on model | – |
| MVA | mva | Product-form exact mean-value analysis (MVA) | [24], [6] |
| MVA | aba.upper | Asymptotic bound analysis (upper bounds) | [5, §9.4] |
| MVA | aba.lower | Asymptotic bound analysis (lower bounds) | [5, §9.4] |

*Continued on next page*

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

| Solver | Method | Description | Refs. |
|---|---|---|---|
| MVA | gb.upper | Geometric square-root bounds (upper bounds) | [9] |
| MVA | gb.lower | Geometric square-root bounds (lower bounds) | [9] |
| MVA | gig1.allen | Allen-Cunneen formula for the GI/G/1 queue | [5, §6.3.4] |
| MVA | gig1.heyman | Heyman formula for the GI/G/1 queue | – |
| MVA | gig1.kingman | Kingman upper bound for the GI/G/1 queue | [5, §6.3.6] |
| MVA | gig1.klb | Kramer-Langenbach-Belz formula for the GI/G/1 queue | [5, §6.3.4] |
| MVA | gig1.kobayashi | Kobayashi diffusion approximation for the GI/G/1 queue | [5, §10.1.1] |
| MVA | gig1.marchal | Marchal formula for the GI/G/1 queue | [5, §10.1.3] |
| MVA | gigk | Kingman approximation for the GI/G/k queue | |
| MVA | mg1 | Pollaczek–Khinchine formula for the M/G/1 queue | [5, §3.3.1] |
| MVA | mm1 | Exact formula for the M/M/1 queue | [5, §6.2.1] |
| MVA | mmk | Exact formula for the M/M/k queue (Erlang-C) | |
| NC | default | Auto-select best method based on model features | – |
| NC | exact | Alias for the `ca` method. | – |
| NC | ca | Exact convolution algorithm | – |
| NC | imci | Improved Monte carlo integration sampler | [27] |
| NC | le | Logistic asymptotic expansion | [8] |
| NC | ls | Logistic sampling | [8] |
| NC | pana | Panacea asymptotic expansion | [21], [25] |
| NC | mmint | McKenna-Mitra integral (2-station models only) | [21] |
| SSA | default | Alias for the `serial` method | – |
| SSA | serial | CTMC stochastic simulation on a single core | [17] |
| SSA | serial.hash | `serial` with state hashing (slower, less memory) | – |
| SSA | para | Parallel simulations (independent replicas) | – |
| SSA | para.hash | `para` with state hashing (slower, less memory) | – |

### 5.2.1 AUTO

The `SolverAuto` class provides interfaces to the core solution functions (e.g., `getAvg`, ...) that dynamically bind to one of the other solvers implemented in LINE (`CTMC`, `NC`, ...). It is often not possible 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 `SolverAuto` 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 `Network` 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`$(i, r)$ is the maximum number of jobs of class $r$ that can be placed at station $i$.

### 5.2.3 FLUID

This solver is based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [23].

The fluid ODEs are normally solved with the `'NonNegative'` ODE solver option enabled. Four types of ODE solvers are used: *fast* or *accurate*, the former only if `options.iter_tol`$> 10^{-3}$, and *stiff* or *non-stiff*, depending on the value of `options.stiff`. The default choice of solver is stored in the following static functions:

- `Solver.accurateStiffOdeSolver`, set to MATLAB's `ode15s`.
- `Solver.accurateOdeSolver`, set to `ode45`.
- `Solver.fastStiffOdeSolver`, set to `ode23s`.
- `Solver.fastOdeSolver`, set to `ode23`.

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 `Network` 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 (`'jsim'` method), which runs JMT's discrete-event simulator. The alternative method is the JMVA analytical solver (`'jmva'` 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 [18]. 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 is primarily based on the Bard-Schweitzer approximate mean value analysis (AMVA) algorithm (`options.method='default'`), but also offers and implementation of the exact MVA algorithm (`options.method='exact'`). Non-exponential service times in FCFS nodes are treated using a M/G/1-type approximation. Multi-server FCFS is dealt with using a slight modification of the Rolia-Sevcik method [26]. DPS queues are analyzed with a time-scale separation method, so that for an incoming job of class $r$ and weight $w_r$, classes with weight $w_s \geq 5w_r$ are replaced by high-priority classes that are analyzed using the standard MVA priority approximation. Conversely the remaining classes are treated by weighting the queue-length seen upon arrival in class $s \neq r$ by the correction factor $w_s/w_r$.

### 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 hashed. The state space is not generated upfront, but rather 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 four methods: 'serial' (default), 'serial.hash', 'para', and 'para.hash'. The serial methods run on a single core, while the parallel methods run on multicore via MATLAB's spmd command. The 'hash' sub-option requires the solver to maintain in memory a hashed list of the node states, as opposed to the joint state vector for the system. As a result, the memory occupancy is lower, but the simulation tends to become slower on models with nodes that have large state spaces, due to the extra cost for hashing.

## 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

```
>> SolverJMT.supports(model)
ans =
  logical
   1
```

It is possible to programmatically check which solvers are available for a given model as follows

```
>> NetworkSolver.loadAllFeasibleSolvers(model)
ans =
  1x6 cell array
    {1x1 SolverCTMC}    {1x1 SolverJMT}    {1x1 SolverSSA}    {1x1 SolverGen} ...
          {1x1 SolverMAM}    {1x1 SolverMVA}
```

In the example, `SolverMAM` is not feasible for the considered model and therefore not returned. Note that `SolverAuto` is never included in the list returned by this methods since this is a wrapper for other solvers.

### 5.3.2 Class functions

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

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

- `getAvg`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for each station and class.
- `getAvgChain`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for every station and chain.
- `getAvgSys`: 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.
- `getAvgNode`: behaves similarly to `getAvg`, 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.
- `getTranAvg`: returns transient mean queue length, utilization and throughput for every station and chain from a given initial state.

Table 5.2: Solver support for average performance metrics

| | | Network Solver | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Function** | **Regime** | CTMC | FLUID | JMT | MAM | MVA | NC | SSA |
| getAvg | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgChainTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgNode | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgNodeTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgSys | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgSysTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgArvR | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgArvRChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgQLen | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgQLenTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgQLenChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgRespT | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgRespTTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgRespTChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgSysRespT | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgTput | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgTputTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgTputChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgSysTput | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgUtil | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgUtilTable | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getAvgUtilChain | Steady-state | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| getTranAvg | Transient | ✓ | ✓ | ✓ | | | | |

- `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.

Table 5.3: 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 | ✓ | | ✓ | | | ✓ | |
| getTranCdfPassT | Transient | | ✓ | | | | | |
| getTranCdfRespT | Transient | | | ✓ | | | | |
| getTranProb | Transient | ✓ | | | | | | |
| getTranProbAggr | Transient | ✓ | | | | | | |
| getTranProbSys | Transient | ✓ | | | | | | |
| getTranProbSysAggr | Transient | ✓ | | | | | | |
| sample | Transient | | | | | | | ✓ |
| sampleAggr | Transient | | | ✓ | | | | ✓ |
| sampleSys | Transient | | | | | | | ✓ |
| sampleSysAggr | Transient | | | ✓ | | | | ✓ |

- 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 Sink and Source nodes, but due to low accuracy when run on open models this feature is disabled in the current release.

### 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).

Table 5.4: Solver support for `Network` nodes

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

The table primarily refeers to invocation of the `getAvg` 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 5.5: Solver support for scheduling strategies

| | | Network Solver | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Strategy** | **Class** | CTMC | FLUID | JMT | MAM | MVA | NC | SSA |
| FCFS | NP | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| INF | NP | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| RAND | NP | ✓ | | ✓ | | ✓ | ✓ | ✓ |
| SEPT | NP | ✓ | | ✓ | | | | ✓ |
| SJF | NP | | | ✓ | | | | |
| HOL | NPPrio | ✓ | | ✓ | | | | ✓ |
| PS | PR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| DPS | PR | ✓ | ✓ | ✓ | | ✓ | | ✓ |
| GPS | PR | ✓ | | ✓ | | | | ✓ |

### 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.

Table 5.6: Solver support for statistical distributions

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

### 5.3.6 Solver options

Solver options are encoded in LINE in a structure array that is internally passed to the solution algorithms. The global defaults for the solvers can be manually adjusted by editing the `lineDefaults.m` file in the root folder.

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

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

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

```
s = SolverJMT(model,'samples',1e6)
opt = SolverJMT.defaultOptions; opt.samples=1e6; s = 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 4.5).

- `config` (`struct`) this is data structure to pass solver-specific configuration options to customize the execution of particular methods.

- `cutoff` (`integer` $\geq 1$) 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` $\geq 1$) 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`$\geq 1$ 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` $\geq 1$) 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` $\geq 1$) 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.
- `timestamp` (`real interval`) requires the transient solver to produce a solution in the specified temporal range. If the value is set to $[\texttt{Inf}, \texttt{Inf}]$ the solver will only return a steady-state solution. In the case of the `FLUID` solver and in simulation, $[\texttt{Inf}, \texttt{Inf}]$ has the same computational cost of $[0, \texttt{Inf}]$ therefore the latter is used as default.
- `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 5.7: Default values of the LINE solver options and their default assignments

| | Solver default | | | | | | |
|---|---|---|---|---|---|---|---|
| **Option** | MVA | CTMC | FLUID | JMT | MAM | NC | SSA |
| cache | true | true | true | true | true | true | true |
| config | | | | | | | |
| cutoff | | (no default) | | | | | |
| force | false | false | false | false | false | false | false |
| keep | | | | false | | | |
| init_sol | [] | | [] | | | | |
| iter_max | $10^3$ | | 10 | | | | |
| iter_tol | $10^{-6}$ | | $10^{-4}$ | | $10^{-4}$ | | |
| method | 'default' | 'default' | 'default' | 'default' | 'default' | 'default' | 'default' |
| samples | | | | $10^4$ | | | $10^4$ |
| seed | rand | rand | rand | rand | rand | rand | rand |
| stiff | | | true | | | | |
| timespan | | [Inf,Inf] | [0,Inf] | [0,Inf] | | [Inf,Inf] | [0,Inf] |
| tol | | $10^{-4}$ | $10^{-4}$ | | | | |
| verbose | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## 5.4 Solver maintenance

The following best practices can be helpful in maintaining the LINE installation:

- To install a new release of JMT, it is necessary to delete (or overwrite) the `JMT.jar` file under the `'SolverJMT'` folder. This forces LINE to download the latest version of the JMT executable.
- To remove temporary by-products of the JMT solver it is recommended to periodically run the `jmtCleanTempDir` script. This is more important when using the `'keep'` option, which stores on disk the temporary `.jsimg` and `.jsimw` models sent to JMT.

# Chapter 6

# Layered network models

In this chapter, we present the definition of the `LayeredNetwork` class, which encodes the support in LINE for layered queueing networks. These models are extended queueing networks where servers, in order to process jobs, can issue synchronous and asynchronous calls among each other. The topology of call dependencies makes it possible to partition the model into a set of layers, each consisting of a subset of the resources, which are then iteratively solved in isolation after certain parameter updates until converging to a solution for the entire network.

## 6.1 LayeredNetwork object definition

### 6.1.1 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 service demands required at the underpinning processor, or calls to entries exposed by some remote tasks.

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

```
model = LayeredNetwork('myLayeredModel');
```

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

```
P1 = Processor(model, 'P1', 1, SchedStrategy.PS);
P2 = Processor(model, 'P2', 1, SchedStrategy.PS);
T1 = Task(model, 'T1', 5, SchedStrategy.REF).on(P1);
T2 = Task(model, 'T2', Inf, SchedStrategy.INF).on(P2);
E1 = Entry(model, 'E1').on(T1);
E2 = Entry(model, 'E2').on(T2);
```

Here, 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 number of servers in the underpinning queueing system for `T1`).

Both processors and tasks can be associated to the standard LINE scheduling strategies. For instasnce, `T2` will process incoming requests in parallel according as an infinite server node, . 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 `Network` object.

### 6.1.2 Describing service times of entries

The service demands placed by an entry on the underpinning processor 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 of 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 final activity, which upon completion terminates the execution of the entry.

For example, we can associate an activity to each entry as follows:

```
A1 = Activity(model, 'A1', Exp(1.0)).on(T1).boundTo(E1).synchCall(E2,3.5);
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 service time on the processor underneath `T2` given by an exponential distribution with rate $\lambda = 2.0$.

At present, LINE 2.0.3 supports both synchronous and partially asynchronous calls, the latter only via SOLVERLQNS.

#### Activity graphs

Often, it is useful to structure the sequence of activities carried out by an entry in a graph. Currently, LINE supports this feature only for activities places in series. For example, we may replace the specification of the activities underpinning a call to `E2` as

```
A20 = Activity(model, 'A20', Exp(1.0)).on(T2).boundTo(E2);
A21 = Activity(model, 'A21', Erlang.fitMeanAndOrder(1.0,2)).on(T2);
A22 = Activity(model, 'A22', Exp(1.0)).on(T2).repliesTo(E2);
T2.addPrecedence(ActivityPrecedence.Serial(A20, A21, A22));
```

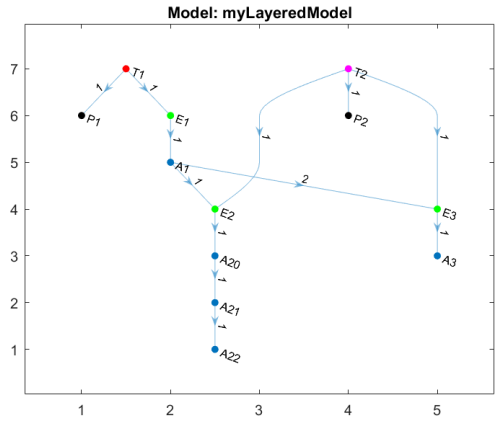


Figure 6.1: `LayeredNetwork.plot` method

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).

### 6.1.3 Debugging and visualization

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

```
plot(model)
```

An example of the result is shown in the next figure. The figure shows two processors (`P1` and `P2`), two tasks (`T1` and `T2`), and three entries (`E1`, `E2`, and `E3`) with their associated activities. Both dependencies and calls are both shown as directed arcs, with the edge weight on call arcs corresponding to the average number of calls to the target entry. For example, `A1` calls `E3` on average 2.0 times. As in the case of the `Network` class, the `getGraph` method can be called to inspect the structure of the `LayeredNetwork` object.

Lastly, 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 cell array 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.

## 6.2 Decomposition into layers

Layers are a form of decomposition where the influence of resources not explicitly represented in that layer is taken into account through an artificial delay station, placed in a closed loop to the resources [26]. This

artificial delay is used to model the inter-arrival time between calls from resources that belong to other layers.

### 6.2.1 Running a decomposition

The current version of LINE adopts SRVN-type layering [15], whereby a layer corresponds to one and only one resource, either a processor or a task. The only exception are reference tasks, which can only appear as clients to their processors. The `getLayers` method returns a cell array consisting of the `Network` objects corresponding to each layer

```
layers = model.getLayers()
```

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., `'AS1=>E3'` is a closed class used represent synchronous calls from activity `AS1` to entry `E3`. 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.

### 6.2.2 Initialization and update

In general, the parameters of a layer will depend on the steady-state solution of an other layer, causing a cyclic dependence that can be broken only after the model is analyzed by a solver. In order to assign parameters within each layer prior to its solution, the `LayeredNetwork` class uses the `initDefault` method, which sets the value of the artificial delay to simple operational analysis bounds [20].

The layer parameterization depends on a subset of performance indexes stored in a `param` structure array within the `LayeredNetwork` class. After initialization, it is possible to update the layer parameterization for example as follows

```
layers = model.getLayers();
for l=1:model.getNumberOfLayers()
    AvgTableByLayer{l} = SolverMVA(layers{l}).getAvgTable;
end
model.updateParam(AvgTableByLayer);
model.refreshLayers;
```

Here, the `refreshParam` method updates the `param` structure array from a cell array of steady-state solutions for the `Network` objects in each layer. Subsequently, the `refreshLayers` method enacts the new parameterization across the `Network` objects in each layer.

## 6.3 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 [15]. 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 `getAvgTable` can be used to obtain a full set of mean performance indexes for the model, e.g.,

```
>> AvgTable = SolverLQNS(model).getAvgTable
AvgTable =
  8x6 table
    Node      NodeType        QLen         Util       RespT       Tput
    ____    ____________    ________    _________    ______    _________
    'P1'    'Processor'        NaN      0.071429     NaN          NaN
    'T1'    'Task'          0.28571     0.071429     NaN       0.071429
    'E1'    'Entry'         0.28571     0.071429       4       0.071429
    'A1'    'Activity'      0.28571     0.071429       4       0.071429
    'P2'    'Processor'        NaN       0.21429     NaN          NaN
    'T2'    'Task'          0.21429     0.21429     NaN        0.21429
    'E2'    'Entry'         0.21429     0.21429       1        0.21429
    'A2'    'Activity'      0.21429     0.21429       1        0.21429
```

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 `getAvgTable` method in `Network` 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.

### 6.3.1 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.

### 6.3.2 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 `Network` solvers described in the solvers chapter to the analysis of individual layers, as illustrated in the following example for the `MVA` solver

```
options = SolverLN.defaultOptions;
mvaopt = SolverMVA.defaultOptions;
SolverLN(model, @(layer) SolverMVA(layer, mvaopt), options).getAvgTable
```

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`.

## 6.4 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.,

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

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

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

[1] https://raw.githubusercontent.com/layeredqueuing/V5/master/xml/lqn.xsd

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.

[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 2.0.0 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 [10].

## 7.1 Environment object definition

### 7.1.1 Specifying the environment transitions

To specify an environment, it is sufficient to define a cell array with entries describing the distribution of time before the environment jumps to a given target state. For example

```
env{1,1} = Exp(0);
env{1,2} = Exp(1);
env{2,1} = Exp(1);
env{2,2} = Exp(0);
```

describes an environment consisting of two stage, where the time before a transition to the other stage is exponential with unit rate. If we were to set instead

```
env{2,2} = Erlang.fitMeanAndOrder(1,2);
```

this would cause a race condition between two distributions in stage two: the exponential transition back to stage 1, and the Erlang-2 distributed transition with unit rate that remains in stage 2. The latter means that periodically the system will be re-initialized in stage 2, meaning that jobs in execution at a server are required all to restart execution.

In LINE, an environment is internally described by a Markov renewal process (MRP) with transition times belonging to the `PhaseType` 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 can be easily transformed into an equivalent continuous-time Markov chain (CTMC) to enable analysis, a task that LINE performs automatically. In the example above, 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.

### 7.1.2 Specifying the system models

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 `Network` 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. An example of this specification style is given in `example_randomEnvironment_1.m` under LINE's example folder.

## 7.2 Solvers

The steady-state analysis of a system in a random environment is carried out in LINE using the blending method [10], 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 `Network` 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 `getAvg` method on the `SolverEnv` object, e.g.,

```
models = {model1, model2, model3, model4};
envSolver = SolverEnv(models, env, @SolverFluid,options);
[QN,UN,TN] = envSolver.getAvg()
```

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] Simonetta 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.

[2] 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.

[3] Dario Bini, Beatrice Meini, S. Steffé, Juan F. Pérez, and Benny Van Houdt. Smcsolver and q-mam: tools for matrix-analytic methods. *SIGMETRICS Performance Evaluation Review*, 39(4):46, 2012.

[4] Andrea Bobbio, András Horváth, Marco Scarpa, and Miklós 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] 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.

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

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

[9] Giuliano 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.

[10] Giuliano Casale, Mirco Tribastone, and Peter G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

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

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

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

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

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

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

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

[18] 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).

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

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

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

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

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

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

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

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

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

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

[29] 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.

# Appendix A

# Examples

The table below lists the scripts available under the `examples` folder.

Table A.1: Examples

| Example | Problem |
|---|---|
| example_cdfRespT_1 | Station response time distribution in a single-class single-job closed network |
| example_cdfRespT_2 | Station response time distribution in a multi-chain closed network |
| example_cdfRespT_3 | Station response time distribution in a multi-chain open network |
| example_cdfRespT_4 | Simulation-based station response time distribution analysis |
| example_cdfRespT_5 | Station response time distribution under increasing job populations |
| example_closedModel_1 | Solving a single-class exponential closed queueing network |
| example_closedModel_2 | Solving a closed queueing network with a multi-class FCFS station |
| example_closedModel_3 | Solving exactly a multi-chain product-form closed queueing network |
| example_closedModel_4 | Local state space generation for a station in a closed network |
| example_closedModel_5 | 1-line exact MVA solution of a cyclic network of PS and INF stations |
| example_closedModel_6 | Closed network with round robin scheduling |
| example_initState_1 | Specifying an initial state and prior in a single class model. |
| example_initState_2 | Specifying an initial state and prior in a multiclass model. |
| example_layeredModel_1 | Analyze a layered network specified in a LQNS XML file |
| example_layeredModel_2 | Specifying and solving a basic layered network |
| example_misc_1 | Use of performance indexes handles |
| example_misc_2 | Update and refresh of service times |
| example_misc_3 | Parameterization of a discriminatory processor sharing (DPS) station |
| example_misc_4 | Automatic detection of solvers that cannot analyze the model |
| example_forkJoin_1 | A single class open fork-join network |
| example_forkJoin_2 | A multiclass open fork-join network |
| example_forkJoin_3 | A closed model with nested forks and joins |
| example_forkJoin_4 | An open model with a fork but without a join |
| example_mixedModel_1 | Solving a queueing network model with both closed and open classes |
| example_openModel_1 | Solving a queueing network model with open classes, scalar cutoff options |
| example_openModel_2 | 1-line solution of a tandem network of PS and INF stations |
| example_openModel_3 | Solving a queueing network model with open classes, matrix cutoff options |
| example_openModel_4 | Trace-driven simulation of an M/M/1 queue |
| example_randomEnvironment_1 | Solving a model in a 2-stage random environment |
| example_randomEnvironment_2 | Solving a model in a 4-stage random environment |
| example_stateProbabilities_1 | Computing marginal state probabilities for a node |
| example_stateProbabilities_2 | Computing marginal state probabilities for a node under class-switching |
| example_stateProbabilities_3 | Computing joint state probabilities for the system |