LINE: Performance and Reliability Analysis Engine

User manual

Version: 2.0.0

Last revision: July 27, 2018

Contents

Chapter 1

Introduction

1.1 What is LINE?

LINE is a tool for performance and reliability analysis of complex systems based on queueing theory and stochastic modeling. Systems may either be software applications, business processes, queueing networks, or else. LINE transforms a high-level system model into one or more stochastic models, typically extended queueing networks, that are subsequently analyzed using either numerical algorithms or simulation. Currently, LINE is able to load and analyze models specified in the following high-level languages:

- *LINE modeling language*. This a MATLAB-based object-oriented language designed to closely resemble the abstractions available in the Java Modelling Tools (JMT) queueing network simulator[1]. Among the benefit of this language is the possibility to transform back-and-forth LINE models into JMT simulation models. Such models can be easily inspected either within MATLAB or using the JSIMgraph tool within JMT.

- *Layered queueing network models (LQN)*. LINE is able to solve a sub-class of LQN models, provided that they are specified using the XML metamodel of the LQNS solver[2].

- *Performance Model Interchange Format (PMIF)*. LINE is able to import and solve closed queueing network model specified using PMIF v1.0.

This document contains installation and usage instructions for the features in version 2.0.0.

1.2 LINE distributions

LINE 2.0.0 can be downloaded at: https://github.com/line-solver/line/releases
Contrary to earlier versions, LINE 2.0.0 is released only as a MATLAB binary (compiled .p files). The distribution has been tested on MATLAB R2018a. If you are interested to obtain a JAR or executable distribution for one of the operating systems supported by the MATLAB Compiler Runtime (MCR), please contact the maintainer.

[1] http://jmt.sf.net
[2] https://github.com/layeredqueuing/

1.3 References

To cite LINE, we recommend to reference:

- J. F. Pérez and G. Casale. "LINE: Evaluating Software Applications in Unreliable Environments", in *IEEE Transactions on Reliability*, 2017.

The following papers discuss recent applications of LINE:

- 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 software engineering*

1.4 Installing LINE

This is the quickest way to getting started with LINE if you have a MATLAB distribution and a basic understanding of MATLAB.

1. Download the desired release from https://github.com/line-solver/line/releases
2. Unzip the file in the desired installation folder.
3. Start MATLAB and use the change the active directory to the installation folder. Then add all LINE folders to the path

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

4. Run the LINE demo code using

```
allExamples
```

1.5 Contact

For issues or feature requests, please get in touch with the LINE team via github:

https://github.com/line-solver/line/issues

Project coordinator and maintainer: Dr. Giuliano Casale, Imperial College London.

1.6 Copyright and license

1.7 Acknowledgement

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

Chapter 2

Network models

LINE can map a system description into one of two main types of stochastic models: `Network` and `LayeredNetwork` models, each specified in a corresponding MATLAB class. `Networks` are extended queueing networks with open or closed classes of moving units (jobs), while a `LayeredNetwork` consists of layers, each corresponding to a queueing network, which interact through synchronous and asynchronous calls.

Throughout this chapter, we discuss the specification of `Network` models, we point to Chapter 4 for `LayeredNetwork` models. It is important to keep in mind that not all of the model features are supported by all LINE solvers. However, upon invoking a solver LINE automatically detects if the supplied model can be analyzed by the chosen solver. If this is not possible, an empty result set will be returned. We point to Chapter 3 for tables summarizing the features supported by each solver.

2.1 Network object definition

2.1.1 Creating a network of resources

A queueing network model can be easily described using the LINE modelling language, which decouples network specification from its solution. To get started, we can create our first queueing network model, called `'myModel'`, as follows

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

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

A *node* is a resource in the network that can be visited by a job. If jobs are allowed to spend time in the node during the visit to the system, the node is said to be a *station*. For example, a first-come first-served queue is a station, while a sink or class-switching router are nodes. Stations are instances of the `Station` class and its descendants. In particular, the `QueueingStation` class specifies a queueing system from its name and scheduling strategy, e.g.

```
node{1} = QueueingStation(model, 'Queue1', SchedStrategy.FCFS);
```

Scheduling strategies are available under 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`)

Infinite-server stations may be equivalently instantiated as stations with the `SchedStrategy.INF` strategy or using the following specialized constructor

```
node{2} = DelayStation(model, 'ThinkTime');
```

If a strategy requires class weights, these can be specified as an argument to the `setService` method, which is discussed later in Section 2.1.5.

2.1.2 Service classes

Jobs travel within the network placing service demands at the stations. 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 nodes. The demand placed by a job at a station depends on the class the job currently belongs to.

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

```
jobclass{1} = OpenClass(model, 'Class1');
node{3} = Source(model, 'Source');
node{4} = Sink(model, 'Sink');
```

Sources are special stations holding an infinite pool of jobs, representing the external world. Sinks are nodes that route a departing job back into this infinite pool. A network can include at most a single source and a single sink. LINE does not require to associate these nodes explicitly with the open classes, as this is done automatically. However, the LINE language requires to explicitly create these nodes as the routing topology needs to specify the entry and exit points of jobs.

To create a closed class, we need instead to indicate the number of jobs initially in the class (e.g., 5 jobs) and the *reference station* (e.g., `node{1}`):

```
jobclass{2} = ClosedClass(model, 'Class2', 5, node{1});
```

If the network includes only closed classes, there is no need to instantiate source and sink.

Reference station

As discussed later, each closed class r belongs to one and only one chain, which defines the set of reachable classes for a job that starts in class r and over time switches class according to the class switching mechanism described later in Section 2.1.3. The reference station is a node used to calculate performance indexes of the closed chains. For example, the system throughput for a chain is defined as the arrival rate of that chain at the reference station. If there is no class switching, each chain contain a single class, thus the system throughput per chain and per class are identical.

Class priorities

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

```
jobclass{2} = ClosedClass(model, 'Class2', 5, node{1}, 0);
```

2.1.3 Routing policies

Jobs travel between nodes according to the network topology and a routing policy. The function `addLink` allows us to specify that two nodes are connected

```
model.addLink(node{1}, node{1}); %self-loop
model.addLink(node{1}, node{2});
```

Typically, a queueing network will use a probabilistic routing policy. If two nodes are connected, it is possible to specify the routing probability of jobs between the jobs for each class, e.g.,

```
node{1}.setProbRouting(jobclass{1}, node{1}, 0.6)
node{1}.setProbRouting(jobclass{1}, node{2}, 0.4)
```

However, the simplest way to specify a large routing topology is to define the routing probability matrix for each class, followed by a method call to the `linkNetwork` method. For example, consider a network with two classes and four nodes where: (i) class one visits node 1, then one at random between nodes 2,3, and 4, and finally cycles back to node 1; (ii) class two cycles among stations 1 to 4 in a sequence. We can instantiate this routing topology as follows:

```
P{1} = zeros(4); P{1}(1,2:4) = [1/3, 1/3, 1/3]; P{1}(2:4,1) = 1.0; % class 1
P{2} = circul(4); % class 2 visits stations in order
model.linkNetwork(P);
```

It is important to note that the routing matrix is specified between *nodes*, instead than between stations, which means that elements such as the sink are explicitly considered in the routing matrix.

The numbering of classes and nodes corresponds to the order in which these elements are instantiated in the model. The `getClassIndex` and `getNodeIndex` methods return the numeric index associated to a name, e.g., `model.getNodeIndex('Delay')`. Class and node names in a network need to be unique and can be fully listed using the `getClassNames` and `getNodeNames` methods.

Class switching

In LINE, jobs can switch class while they travel between stations (including self-loops on the same station). This allows to model queueing properties such as re-entrant lines, whereby a job visiting a station a second time may require a different average demand than at the first visit. Jobs in open classes can switch only to another open class. Similarly, jobs in closed classes can only switch to a closed class. Class switching from open to closed classes (or viceversa) is forbidden. The policy to describe the class switching process is integrated in the specification of the routing between stations.

In models with class switching, a routing matrix is required for each possible pair of source and target classes. For example, assume that in the previous example a job in class 1 can switch into class 2 after departing node 4, remaining there forever. We can specify this routing policy as follows:

```
P = cellzeros(2,4); % 2 classes, 4 stations
% switching probabilities from class 1 to class 2
P{1,2}(4,1) = 1.0; % routing from station 4 to 1
% routing probabilities for class 1 (no switching)
P{1,1}(1,2:4) = [1/3, 1/3, 1/3]; P{1,1}(2:3,1) = 1.0;
% routing probabilities for class 2 (no switching)
P{2,2} = circul(4); % class 2 visits stations in order
model.linkNetwork(P);
```

In the example, `P{r,s}` is the routing matrix for jobs switching from class `r` to `s`, e.g., `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`.

Depending on the specified routing topology, a job will be able to switch among a subset of the available classes. Each subset is called a *chain*. Chains correspond to weakly connected components of the routing probability matrix of the network, when this is seen as an undirected graph. The function `model.getChains` produces a list of chains with their member classes. LINE assumes that a job switches class immediately *before* leaving a station, so that the throughput of the station and the routing probabilities are already accounting for the new class of the job.

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.

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

```
D = [11,12; 21,22]; % D(i,r) - class-r demand at station i (PS)
A = [10,20]; % A(r) - arrival rate of class r
Z = [91,92; 93,94]; % Z(i,r)  - class-r demand at station i (INF)
modelPsInf = Network.tandemPsInf(D,A,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` methods provide static constructors for networks with other combinations of scheduling policies.

If a tandem network is visited only by closed classes it is then called a cyclic network. Similar to tandem networks, LINE offers a set of static constructors: `Network.cyclicPs`, `Network.cyclicPsInf`, `Network.cyclicFcfs`, and `Network.cyclicFcfsInf`. These methods only require to replace the arrival rate vector `A` by a vector specifying the job populations for each of the closed classes.

2.1.4 Finite buffers

The station methods `setBufferCapacity` and `setBufferChainCapacity` are used to place constraints on the number of jobs, total or for each chain, that can reside at a station. Some of the LINE solvers are able to take such a constraints into account. Note that LINE does not allow one to specify buffer constraints at the level of individual classes, unless they belong to separate chains, in which case `setBufferChainCapacity` is sufficient for the purpose. For example,

```
example_closedModel_3
node{2}.setBufferChainCapacity([1,1])
model.refreshBuffer()
```

creates an example model with two chains and three classes (specified in `example_closedModel_3`) and then 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 `setBufferChainCapacity([1,7])`, which exceeds the total job population in chain 2, LINE would have automatically reduced this value to the job population. This ensures that methods that analyze the state space of the model do not consider unreachable states.

The `refreshBuffer` method updates the model after the change. The latter is required because LINE, after first invocation of the solvers, uses lazy updates of the model parameters. Since `example_closedModel_3.m` has already invoked a solver, new changes are materially applied by LINE to the network only after calling an appropriate `refresh` method, see Section 2.4. If the buffer capacity changes were made before the first solver invocation, then there would not be need for a `refreshBuffer` call.

2.1.5 Service and inter-arrival time processes

A number of statistical distributions are available for job service times at the stations and inter-arrival times from the `Source`. The class `MarkovianDistribution` offers distributions that are analytically tractable, which are defined upon certain absorbing Markov chains consisting of one or more states (*phases*). They include the following distributions with the respective constructors:

- Exponential distribution: `Exp`(λ), where λ is the rate of the exponential
- n-phase Erlang distribution: `Erlang`(α, n), where α is the rate of each of the n exponential phases
- 2-phase hyper-exponential distribution: `HyperExp`$(p, \lambda_1, \lambda_2)$, that returns an exponential with rate λ_1 with probability p, and an exponential with rate λ_2 otherwise.
- 2-phase Coxian distribution: `Cox2`(μ_1, μ_2, ϕ_1), which assigns rates μ_1 and μ_2 to the two rates, and completion probability from phase 1 equal to ϕ_1 (the probability from phase 2 is $\phi_2 = 1.0$).

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

```
node{1}.setService(jobclass{2}, Cox2.fitMoments(0.2,10));
```

Inter-arrival time distributions can be instantiated in a similar way, using `setArrival` instead of `setService` on the `Source` node.

Non-Markovian distributions are also available, but typically they restrict the available network analysis techniques to simulation. They include the following distributions:

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

Lastly, we discuss two special distributions. The `Disabled` distribution allows us to explicitly forbid a class to receive service at a station. Performance metrics for disabled classes will be set to `NaN`. Conversely, the `Immediate` class is treated as instantaneous service (zero service time). Typically, LINE solvers will replace zero service times with small positive values ($\varepsilon = 10^{-7}$).

Fitting a distribution

The `fitMoments` method is available for all distribution inheriting from `MarkovianDistribution` and provides exact or approximate matching of the requested moments, depending on the theoretical constraints imposed by the assumed form of the distribution. For example, an Erlang distribution with SCV=0.75 does not exist, as the n-phase Erlang has SCV=$1/n$. In a case like this, `Erlang.fitMoments(1,0.75)` returns a 2-phase Erlang (SCV=0.5) with unit mean. The Erlang distribution also offer a method `fitMeanAndOrder`(μ, n), which instantiates a n-phase Erlang with the given mean μ.

In distributions that are uniquely determined by more than two moments, `fitMoments` 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.fitMoments` chooses to return a probability of selecting phase 1 equal to 0.99, as this spends the degree of freedom corresponding to the (unspecified) third moment of the distribution.

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` methods, e.g.,

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

both return 1. Moreover, the `sample` method can be used to generate values from the obtained distribution, e.g. we can generate 3 samples as

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

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

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

Lastly, for the distributions of the `MarkovianDistribution` class it is possible to obtain the standard (D_0, D_1) representation used in the theory of Markovian arrival processes by means of the `e.getMarkovianRepresentation` method.

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 (`Trace`). LINE assumes that empirical traces are supplied as text files (ASCII), formatted as a column of numbers.

Scheduling parameters

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

```
node{1}.setService(jobclass{2}, Cox2.fitMoments(0.2,10), 5.0);
```

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

2.1.6 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 2.1. Using a related function, `jsimwView`, it is also possible to export the model to the JSIMwiz environment, which offers a wizard-based interface.

Another way to debug a LINE model is to transforming it into a MATLAB graph object, e.g.

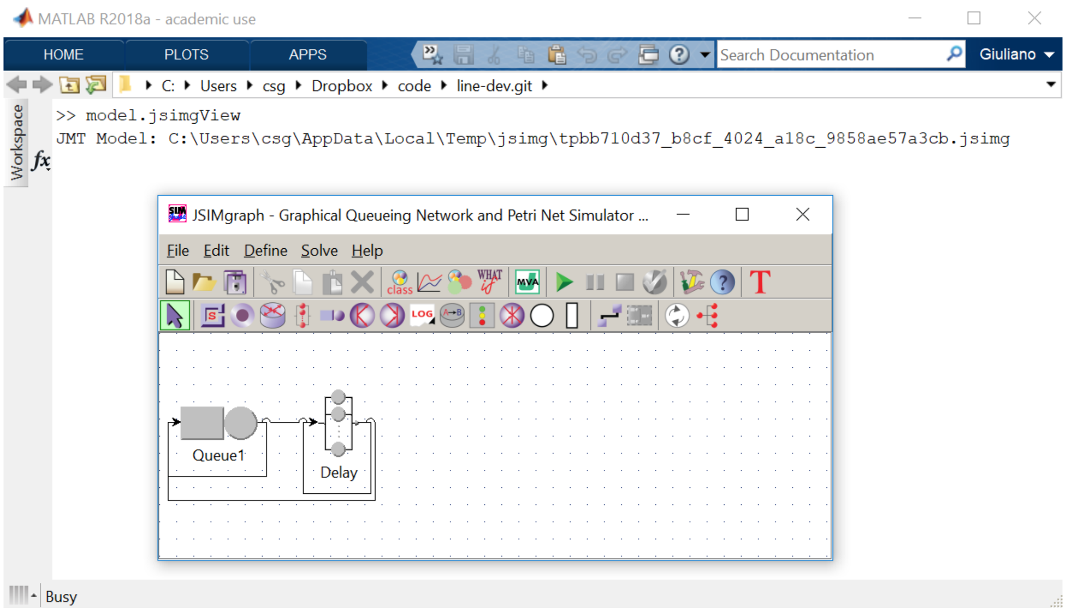
```
G = model.getGraph();
plot(G,'EdgeLabel',G.Edges.Weight,'Layout','Layered')
```

plots a graph of the queueing network topology. Furthermore, the graph properties concisely summarize the key features of the network

```
>> G.Nodes
ans =
  2x5 table
      Name             Type              Sched     Jobs     ClosedClass1

    ________    _________________    _____    ____    ____________

    'Delay'     'DelayStation'       'inf'     5             1
    'Queue1'    'QueueingStation'    'ps'      0             2
>> G.Edges
ans =
  3x4 table
          EndNodes            Weight     Rate         Class
```



```
>> model.jsimgView
JMT Model: C:\Users\csg\AppData\Local\Temp\jsimg\tpbb710d37_b8cf_4024_a18c_9858ae57a3cb.jsimg
```

MATLAB R2018a - academic use
HOME
PLOTS
APPS
Search Documentation
Giuliano
C: ▸ Users ▸ csg ▸ Dropbox ▸ code ▸ line-dev.git ▸
Workspace
JSIMgraph - Graphical Queueing Network and Petri Net Simulator ...
File Edit Define Solve Help
Queue1
Delay
Busy

Figure 2.1: `Network.jsimgView` method

```
    ____________    ____________    _____    ____    ______________
    '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 source node.

2.1.7 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');
node{1} = DelayStation(model, 'Delay');
```

```
node{2} = QueueingStation(model, 'Queue1', SchedStrategy.PS);
node{2}.setNumServers(1);
jobclass{1} = ClosedClass(model, 'ClosedClass1', 5, node{1}, 0);
node{1}.setService(jobclass{1}, Cox2.fitMoments(1.000000,1.000000));
node{2}.setService(jobclass{1}, Cox2.fitMoments(2.000000,1.000000));
myP = cell(1);
myP{1,1} = [0.7 0.3;1 0];
model.linkNetwork(myP);
```

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

2.2 Defining network 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 native support in LINE for state modeling.

2.2.1 Station states

We begin by explaining how to specify a state s_0. State modelling is supported only for stations with scheduling policies that depend on the number of jobs running or waiting at the node. For example, it is not supported for shortest job first (`SchedStrategy.SJF`) scheduling, in which state depends on the service time samples for the jobs.

Suppose that the network has R classes and that service distributions are phase-type, i.e., that they inherit from `MarkovianDistribution`. 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
- 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 `MarkovianDistribution`. If the distribution is not Markovian, then there is a single phase. With these definitions, Table 2.1 illustrates how to specify in LINE a valid state for a station depending on its scheduling policy. States

Table 2.1: State descriptors for Markovian scheduling policies

Strategy	**Station state**
`FCFS`, `HOL`, `LCFS`	$[c_b, c_{b-1}, \ldots, c_1, s_{11}, \ldots, s_{1K_1}, s_{21}, \ldots, s_{2K_2}, \ldots, s_{R1}, \ldots, s_{RK_R}]$
`SEPT`, `RAND`	$[n_1, n_2, \ldots, n_R, s_{11}, \ldots, s_{1K_1}, s_{21}, \ldots, s_{2K_2}, \ldots, s_{R1}, \ldots, s_{RK_R}]$
`PS`, `DPS`, `GPS`, `INF`	$[s_{11}, \ldots, s_{1K_1}, s_{21}, \ldots, s_{2K_2}, \ldots, s_{R1}, \ldots, s_{RK_R}]$

can be manually specified or enumerated automatically through LINE. Library functions for handling states are as follows:

- `State.fromMarginal`: enumerates all states that have the same marginal state $[n_1, n_2, \ldots, n_R]$.
- `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 of resident jobs.
- `State.toMarginal`: extracts 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,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 wether 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 = stateFromMarginalAndStarted(model,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.

Assigning a state to a station

Given a single or multiple states, it is possible to assign the initial state to a station using the `setState` method on that station's object. To cope with multiple states, 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 analyze the network from all those states and weight the results according to the prior probabilities. The default prior value assigned probability 1.0 to the *first* specified state. The station methods `setStatePrior` and `getStatePrior` can be used to check and change the prior probabilities for the supplied initial states.

2.2.2 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 methods supplied by the `Network` class. These methods 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` method 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 server of reference stations are occupied in order of class id, i.e., jobs in the firstly created class are assigned to the server in phase 1, then spare servers are allocated to the second class in phase 1, and so forth;
- if the scheduling policy 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.

2.2.3 Station response time distribution

`SolverFluid` supports the computation of response time distributions for individual classes through the `getCdfRespT` method. The method 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)');
```

2.2.4 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 methods `getAvgSysByChain` and `getAvgSysByChainTable` 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 in some applications one may want to avoid counting each visit as a completion of the visit to the system. In such cases, LINE allows 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.

```
jobclass{1} = ClosedClass(model, 'ClosedClass1', 1, node{1}, 0);
jobclass{2} = ClosedClass(model, 'ClosedClass2', 0, node{1}, 0);

jobclass{1}.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.linkNetwork(P);
```

Note that LINE does not allow a chain to complete at heterogeneous stations, therefore the `completes` property of a class always refers to the reference station for the chain.

2.3 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 $E[n(t)|s_0]$.

2.3.1 Computing transient averages

At present, LINE supports only transient computation of queue-lengths, throughputs and utilizations using the `CTMC` and `FLUID` solvers. Transient response times are not currently supported, as they do not always obey Little's law.

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

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

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

```
[QNt,UNt,TNt] = solver{s}.getTransientAvg(Qt,Ut,Tt)
plot(QNt{1,1}(:,2), QNt{1,1}(:,1))
```

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

Note that the above code does not show how to specify a maximum time t for the output time series. This can be done using the `timespan` field of the options, as described later in Chapter 3.

2.3.2 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. To avoid ambiguities, the notion of response time distribution is best replaced 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 method `getCdfFirstPassT` provides this distribution, assuming as initial state the one specified for the model, e.g., using `setState` or `initDefault`. This method is available only in `SolverFluid` and has a similar syntax as `getCdfRespT`.

2.4 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.

2.4.1 Internal representation

For efficiency reasons, once a user requests to solve a `Network`, LINE calls internally generates a static representation of the network using the `generateQN` method, and then passes this object to the chosen solver to parameterize the analysis.

The main representation used within LINE is the `QNCoxCS` class, which describes an extended multiclass queueing network with class-switching and Coxian service times. The representation can be obtained as follows

```
  myQN = model.getQN()
```

Table 2.2 presents the properties of the `QNCoxCS` class.

2.4.2 Fast parameter update

Successive invocations of `getQN` will return a cached copy of the `QN` representation, unless the user has called `model.refresh()` or `model.reset()` in-between the invocations. The `refresh` method 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 `refresh` or `getQN` call.

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

Table 2.2: `QNCoxCS` class properties

Field	Type	Description
`bufsz`(i)	`integer`	Total buffer size at station i
`bufszclass`(i, r)	`integer`	Maximum buffer size used by class r at station i
`chains`(c, r)	`logical`	`true` if class r is in chain c, or `false` otherwise
`classname`$\{r\}$	`string`	Name of class r
`classprio`(r)	`integer`	Priority of class r (0 = highest priority)
`mu`$\{i, r\}(k)$	`double`	Coxian service or arrival rate in phase k for class r at station i
`nchains`	`integer`	Number of chains in the network
`nchainjobs`(c)	`integer`	Number of jobs in chain c
`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 in station i
`nstations`	`integer`	Number of stations in the network
`phi`$\{i, r\}(k)$	`double`	Coxian completion probability in phase k for class r at station i
`phases`(i, r)	`integer`	Number of phases for service process of 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`	Routing probability from station i to j, switching class from r to s where, e.g., $idx_{ir} = (i-1) *$ `myQN.nclasses` $+ r$.
`sched`$\{i\}$	`cell`	Scheduling strategy at station i (e.g., `SchedStrategy.PS`)
`schedparam`(i, r)	`double`	Parameter for class r strategy at station i
`schedid`(i)	`integer`	Scheduling strategy id at station i (e.g., `SchedStrategy.ID_PS`)
`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 vector`	The t-th state in the state space (or a portion thereof). This field is initially empty and assigned by the solver during execution.
`state`	`integer vector`	Initial state of the model. This field is initially empty and assigned by the solver during execution.
`stationname`$\{i\}$	`string`	Name of station i
`visits`$\{c\}(i, r)$	`double`	Number of visits that a chain c job pays at node i in class r

- `refreshArrival`: this method should be called after updating the inter-arrival distribution at a `Source`.
- `refreshBuffer`: this method should be called after changing buffer capacities, as it updates the `bufsz` and `bufszclass` fields.
- `refreshChains`: this method should be used after changing the routing topology, as it refreshes the `rt`, `chains`, `nchains`, `nchainjobs`, and `visits` fields.
- `refreshPriorities`: this method updates class priorities by refreshing 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:

```
node{1}.setService(jobclass{1}, Exp(1.0));
model.refreshService;
```

2.4.3 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` method should be used to clone LINE objects, for example before modifying a parameter for a sensitivity analysis. This method 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` method 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 3

Network solvers

3.1 Overview

Solvers analyze objects of class `Network` to return average, transient, distributions, or state probability metrics.

Available solvers

The following `Network` solvers are available within LINE 2.0.0:

- `AMVA`: This is a solver based on approximate mean-value analysis (AMVA). 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 `SolverAMVA` 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 [8, 9]. 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 model [1]. This 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.
- `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 simulation based on the CTMC representation of the model. It is fully implemented in MATLAB language and thus offer lower speed than JMT, but the model execution can be easily parallelized using MATLAB's *spmd* construct. This solver is implemented by the `SolverSSA` class.

3.2 Solver features and options

Scheduling strategies

Table 3.1 shows the supported scheduling strategies within LINE queueing stations. Each strategy belongs to a policy class: preemptive resume (`SchedPolicy.PR`), non-preemptive (`SchedPolicy.NP`), non-preemptive priority (`SchedPolicy.NPPrio`).

Table 3.1: Solver support for scheduling strategies

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

Statistical distributions

Table 3.2 summarizes the current level of support for arrival and service distributions within each solver. `Trace` 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.

Solver options

Solver-specific default options can be obtained by the `defaultOptions` static method of the corresponding class, e.g.,

```
SolverJMT.defaultOptions
```

Unspecified entries indicate that the option is not available. Options are as follows:

- `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.

Table 3.2: Solver support for statistical distributions

	Network Solver						
Distribution	`AMVA`	`CTMC`	`FLUID`	`JMT`	`MAM`	`NC`	`SSA`
Cox2	✓	✓	✓	✓	✓	✓	✓
Exp	✓	✓	✓	✓	✓	✓	✓
Erlang	✓	✓	✓	✓	✓	✓	✓
HyperExp	✓	✓	✓	✓	✓	✓	✓
Disabled	✓	✓	✓	✓	✓	✓	✓
Det				✓			
Gamma				✓			
Pareto				✓			
Trace				✓			
Uniform				✓			

- `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 `AMVA` 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 `AMVA`, this is a matrix where element (i, j) is the mean queue-length at station i in class j. In the case of `FLUID`, this is a model-dependent vector with the values of all the variables used within the ODE system that underpins the fluid approximation.
- `keep` (`logical`) determines if the model-to-model transformations store on file their intermediate outputs. In particular, if `verbose`≥ 1 then the location of the `.jsimg` models sent to JMT will be printed on screen.
- `method` (`solver dependent`) enables to configure the internal algorithms used by a given solver.
- `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 spec-

ified temporal range. If the value is set to [`Inf`, `Inf`] the solver will only return a steady-state solution. In the case of the `FLUID` solver and in simulationHor17, [`Inf`, `Inf`] has the same computational cost of [`0`, `Inf`] therefore the latter is used as default.

- `verbose` controls the verbosity level of the solver. Supported levels are 0 for silent, 1 for standard verbosity, 2 for debugging.

3.3 Network solvers

3.3.1 SolverAMVA class

The solver is based on the Bard-Schweitzer AMVA. Non-exponential service times in FCFS nodes are treated using a diffusion approximation [6]. Multi-server FCFS is dealt with using a slight modification of the Rolia-Sevcik method [11]. 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 AMVA 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.

3.3.2 SolverCTMC class

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 and the ODE solver by the `stiff` option.

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 specify the maximum number of jobs that is allowed in any station.

3.3.3 SolverFluid class

This solver is based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [10].

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 options for the solver are stored in the following static methods:

- `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. When initializing FCFS queues, jobs in the buffer are all initialized in the first phase of the service.

3.3.4 SolverJMT class

The class is a wrapper for the `JMT` simulation and consists of a model-to-model transformation from the `Network` data structure into the JMT's input XML format (`.jsimg`) and a corresponding parser for JMT's results. In the transformation, 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.

3.3.5 SolverMAM class

This is a basic solver for some Markovian open queueing systems that can be analyzed using matrix analytic methods. The solver at the moment is a basic wrapper for the BU tools library for matrix-analytic methods [5]. At present, it is not possible to solve a queueing network model using `SolverMAM`.

3.3.6 SolverSSA class

`SolverSSA` 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` methods of the `Network` class.

3.4 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 the `JMT.jar` file under `'\solvers\JMT'`. This forces LINE to download the latest version of the JMT executable.
- Periodically running the `jmtCleanTempDir` script can help removing temporary by-products of the JMT solver. This is strongly encouraged under repeated uses of the `keep=1` option, as this stores on disk the temporary models sent to JMT.

Chapter 4

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. We point to [3] and to the LQNS user manual for an introduction [4].

4.1 LayeredNetwork object definition

4.1.1 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', 1, 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`). Note that both processors and tasks can be associated to the standard LINE scheduling strategies, with the exception that `SchedStrategy.REF` should be used to denote the reference task, which has a similar meaning to the reference node in the `Network` object.

4.1.2 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 $\lambda = 2.0$.

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));
```

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).

4.1.3 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 Figure 4.1. 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 object.

4.2 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

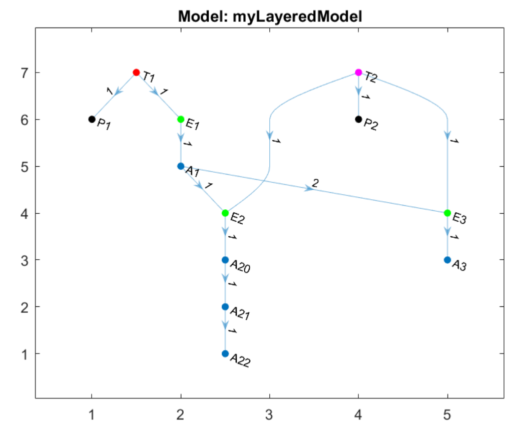


Figure 4.1: `LayeredNetwork.plot` method

resources [11]. This artificial delay is used to model the inter-arrival time between calls from resources that belong to other layers.

4.2.1 Running a decomposition

The current version of LINE adopts SRVN-type layering [4], 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.

4.2.2 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 [7].

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} = SolverAMVA(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.

4.3 Steady-state analysis

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 [4]. 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.

4.3.1 SolverLQNS class

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`).

4.3.2 SolverLN class

The native `LN` solver iteratively applies the updates described in Section 4.2.2 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 Chapter 3 to the analysis of individual layers, as illustrated in the following example for the `AMVA` solver

```
options = SolverLN.defaultOptions;
amvaopt = SolverAMVA.defaultOptions;
SolverLN(model, @(layer) SolverAMVA(layer, amvaopt), 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`.

4.4 Model import and export

A `LayeredNetwork` can be easily read from, or written to, a XML file in the LQNS metamodel format. 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)
```

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

Chapter 5

Random environments

Systems modeled with LINE can be described as operating in an environment whose state affects the way the system operates. The 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 whose input parameters change with the environment stage. For example, an environment with two stages, normal load and peak load, may differ for the number of servers that are configured in a server station, i.e., the user may add servers during peak load. Upon a stage change in the environment, the model parameters will instantaneously change, but the 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 (e.g., the rate at which jobs are served) and the environment dynamic happen at similar timescales [2].

5.1 Environment object definition

5.1.1 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 `MarkovianDistribution` 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.

5.1.2 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.

5.2 Steady-state analysis

The analysis of a system in a random environment is carried out in LINE using the blending method [2], 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.

5.2.1 SolverEnv class

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] 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.

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

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

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

[5] 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).

[6] Hisashi Kobayashi. Application of the diffusion approximation to queueing networks I: equilibrium queue distributions. *J. ACM*, 21(2):316–328, 1974.

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

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

[9] J. F. Pérez and G. Casale. Convergence and estimation with a PS-delay queueing network. Technical report, Imperial College London, 2013.

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

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