LN: A Meta-solver for Layered Queueing Network Analysis

Giuliano Casale(✉), Yicheng Gao, Zifeng Niu, and Lulai Zhu

Department of Computing, Imperial College London, London, UK
{g.casale,y.gao20,zifeng.niu19, lulai.zhu15}@imperial.ac.uk

Abstract. We overview LN, a novel solver introduced in the LINE software package to analyze layered queueing network (LQN) models. The novelty of the LN solver lies in its capability to analyze LQNs with a user-defined combination of solution paradigms, including discrete-event and stochastic simulation, continuous-time Markov chain analysis (CTMC), normalizing constant evaluation (NC), matrix analytic methods (MAM), mean-field approximations (FLUID), and mean-value analysis (MVA). Being parametric in the solver used for each LQN layer, LN as a whole enables the efficient computation of advanced performance metrics such as marginal and joint state probabilities, response and passage time distributions, and transient measures, leveraging individual strengths of the supported solution paradigms. We discuss in particular recent developments added to NC, the default layer solver of LN, which significantly improve the solution of queueing network models obtained using loose layering of the LQN.

Keywords: Layered queueing networks · Computational algorithms · Class switching · Performance measures

1 Introduction

LINE[1] is an open-source software package for analyzing extended queueing network models [9]. The package implements several tens of solution algorithms grouped into solvers, each embodying a specific paradigm for queueing analysis, either simulation-based or analytical (CTMC, NC, MAM, FLUID, MVA). In this paper, we present the LN solver available within the LINE suite version 2.0, which adds a capability to analyze LQNs, a class of extended queueing networks featuring simultaneous resource possession. LN is the first LQN meta-solver, i.e., it offers the flexibility to parametrically choose any of the aforementioned paradigms to evaluate individual layers that compose an LQN. This feature greatly extends the scope of the original LQN solver available in the first version of LINE [28], which was supporting the solution of each layer based on mean-field

[1] http://line-solver.sf.net/.

approximations only. The present paper is the first one to review LQN analysis methods available in the LINE 2.0.x releases.

LINE is open sourced under a permissive BSD-3-Clause license. It is mainly developed in MATLAB, with a few components coded in Java for computational efficiency. A royalty-free Docker image built on the MATLAB compiler runtime is also made available so that end users can run the tool as a service without licensing costs. Some solution methods are implemented based on external solvers that include JMT [2], LQNS [15], BuTools [19], KPC-Toolbox [12], and Q-MAM [3]. In essence, LINE acts as an integration point for multiple queueing analysis tools, providing a common model specification language for their joint use along with its native solvers. For example, an LQN may be analyzed via LQNS for a fast solution, and the result then verified using a slower simulation-based trace-driven execution of the LN solver. This is especially useful in research studies, to detect bugs and compare efficiency of alternative solution methods. JMT is used to visualize models through automated model-to-model transformation and for simulation-based analysis [9]. Besides, LINE is complementary to other efforts to broaden the availability of queueing network algorithms for performance evaluation educators and practitioners, such as Octave queueing [23] and PDQ [18]. Compared to these, LINE adds several advanced algorithms not available in existing software packages.

Related Work. State-of-the-art solvers specific to LQNs include for example LQNS [15] and DiffLQN [33,34]. LQNS is an established solver with a long record of application to real-world software engineering case studies. At heart, the tool applies to the LQN layers approximate mean-value analysis for extended queueing network models [15]. LQN analysis using GreatSPN [1] is also supported via its petrirsrvn tool.

DiffLQN is instead a solver that is based on the mean-field approximation theory developed in the context of PEPA models to scalably analyze LQNs [33,34]. The mean-field fluid paradigm is particularly suited to the solution of large models, as it becomes asymptotically exact in layers with multi-server FCFS stations once the number of jobs and servers grows large in a fixed ratio. Subsequent work on mean-field approximations has further generalized the fluid solutions to processor sharing (PS) stations, class switching, random environments and response time percentiles [28], differentiated service weights [38], multi-class FCFS approximations [9], and mixed models [30].

LN is built around the experience of these LQN solvers, integrating many of the approximate MVA and fluid methods proven effective in the above studies. In addition, it enables the analysis of LQNs using solution paradigms that are uncommon for LQNs, such as continuous-time Markov chains, matrix analytic methods, and normalizing constant evaluation techniques, all of which are not available in existing solvers. As we show later, these paradigms are helpful in computing several LQN metrics that are difficult or impossible to obtain with mean-value analysis or mean-field fluid approximations.

Theoretical Contributions. In developing the LN solver, we have advanced the theory of product-form queueing networks for models consisting of a single infinite-server node and m replicated queueing stations (i.e., having identical service demands in every class). Each station offers multi-class service, according to per-class service time distributions that are possibly load-dependent. We shall refer to such models as *homogeneous layers*, since they naturally arise from a certain LQN decomposition style known as loose layering [16, §3.2], that can be used to analyze arbitrary LQNs. The main theoretical contributions of this paper, are as follows:

- We develop a Gaussian quadrature method for approximating the normalizing constants of homogeneous layers, which leads to a fast computation of their associated performance metrics. By controlling the order of the quadrature, these methods can trade accuracy for speed, while retaining linear worst-case complexity in the total population size.
- We propose a method of moments algorithm for *exactly* solving homogeneous layers in linear time with respect to the total population size when the queueing stations have a single server. We show its ability to handle multi-class models with thousands of jobs in a few milliseconds. As opposed to existing methods such as MoM [7] and CoMoM [6] that can *theoretically* achieve linear complexity, the proposed technique is the first one that realizes this in concrete implementations by avoiding the use of exact arithmetic, which introduces overheads up to about log-linear in the total population size [6,7]. Moreover, it does so without solving systems of linear equations usually appearing in methods of moment algorithms. The result provides efficient approximations for more complex networks with multi-server stations [11,31] and for non-product-form models.
- We derive a related method for marginal probability computations in homogeneous layers featuring quadratic complexity in the total population size. This method also does not require solving systems of linear equations.

We illustrate the application of the above methods to LN and exemplify the other features of the solvers through case studies. In particular, we demonstrate the ability of LN, as a meta-solver, to study performance metrics that cannot be easily analyzed with other LQN solvers, for example integrated models of queueing and caching.

The rest of the paper is organized as follows. Section 2 describes the modeling formalism supported by the LN solver. The overall solution approach and advanced features unique to LN are discussed in Sect. 3. Section 4 elaborates novel solution algorithms offered by the solver. Some case studies are presented in Sect. 5 to illustrate the distinguishing features of LN. Finally, Sect. 6 is dedicated to the conclusions. Proofs of the solution algorithms are given in the Appendix together with an overview of the software architecture of LN.

2 LQN Formalism

LINE offers exact, approximate, asymptotic and simulation-based analysis of open, closed, and mixed multi-class queueing network models. In these models,

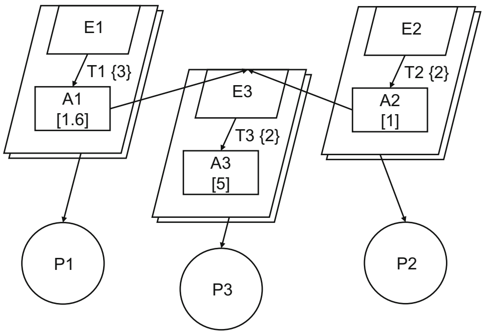


Fig. 1. Example of an LQN model.

jobs are probabilistically routed across a set of nodes, usually queueing stations, where they receive service, typically subject to contention by other jobs. Each job belongs to a class, i.e., a type that defines its service, routing, and arrival characteristics at each node. LINE also supports extensions commonly required in applications such as class switching, non-exponential service times, load dependence, and priorities. The problem is to obtain station and system performance measures such as average queue lengths, utilizations, response times, and throughputs/arrival rates.

Among the most feature-rich extended queueing network models is the class of *layered queueing networks* (LQNs), which has found broad application in software performance engineering [15]. We point the reader to [35] for a comprehensive introduction to LQNs and discuss here only the essential concepts. In an LQN, job visits to the system are modeled as directed acyclic graphs that invoke entries exposed by tasks running on host processors. A workflow of one or more activities (i.e., service phases) is executed at each invocation of an entry. This workflow is called the *activity graph* bound to the entry. Within it, an activity may issue a synchronous call to an entry, while keeping a server in the task blocked, leading to simultaneous resource possession. Asynchronous calls are also possible, which behave similarly to job movements in ordinary queueing networks.

Figure 1 shows an example that contains all the basic elements of LQN models: *tasks*, *host processors*, *entries*, and *activities*. Tasks are depicted as stacked parallelograms and their multiplicities are indicated within curly brackets. A task runs on a single processor (e.g., P1), which is represented by a circle. Specific services provided by a task are called entries and drawn as smaller parallelograms inside that task. Each rectangle denotes a particular activity performed during the execution of an entry. The number between square brackets specifies the service demand of the activity (e.g., 1.6 for activity A1). Workloads in LQNs are generated by a special task termed the reference task, e.g., tasks T1 and T2 in Fig. 1 which model two classes of users with 3 and 2 jobs each and both call entry E3 of task T3.

Extensive prior work in the area has shown that an LQN can be accurately solved by iteratively evaluating a collection of ordinary mixed queueing networks, obtained via decomposition, until reaching a fixed-point solution. Each decomposed sub-model conceptually represents a layer of the client-server system being modeled [29]. These models interact, in the sense that the outputs parameters of one model (e.g., its response times) may form the input parameters of another model (e.g., its service times). In LINE, a collection of interactive models is referred to as an *ensemble*, which is not restricted to LQNs and can encompass other formalisms such as caching models [17]. For this reason, LN may be seen as a general-purpose layered stochastic network solver.

3 LQN Decomposition and Iterative Solution

This section describes the algorithmic methods underpinning the LN solver. We particularly focus on the strategy to divide a given LQN model into multiple layers and the default, though customizable, solution paradigm applied to analyzing each resultant layer.

3.1 Layering Strategy

Prior art has extensively investigated layering strategies, i.e., methods for generating a decomposition of an LQN model into an ensemble of ordinary queueing networks on which solution algorithms can be instantiated. In the current release, the LN solver adopts loose layering [16, §3.2], which ensures that each layer includes replicated queueing stations (i.e., an LQN task or host processor) coupled with an infinite-server node to model the inter-request times of clients. An exception to this rule is that identical replicas of the queueing station are also generated by LN in the same layer. For example, the model in Fig. 1 features under the loose layering style 4 layers: T1→ P1, T2→P2, T3→P3, and (T1,T2)→T3, where → indicates a client-server relationship.

The rationale for choosing loose layering as the default strategy is that, for a total of m queueing stations, many queueing network analysis methods are computationally more efficient in solving m small models with a single (possibly replicated) queueing station rather than a monolithic model comprising m queueing stations. For example, a CTMC solver may be fairly scalable for single queueing systems, but easily incurs exponential state space explosion for queueing networks. A drawback of loose layering is that heterogeneous load balancing or fork/join sections are challenging to model as the participating queueing stations may be scattered across different layers.

Solutions of interactive models in an ensemble are reconciled through fixed-point iterations until performance metrics across the layers are consistent within a predefined numerical tolerance. To this end, the LQNS solver adopts an "elevator" algorithm whereby the graph that describes the client-server relationships is traversed top down and bottom up in an alternate fashion after topological sorting, thus cyclically inverting the order in which the layers are analyzed [16]. The same algorithm is implemented in LN to iterate over all the models within an ensemble.

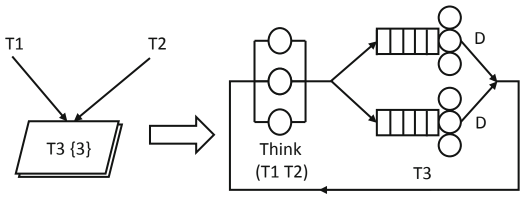


Fig. 2. A layer of the LQN model with replication $m = 2$ and multiplicity $c = 3$.

3.2 Homogeneous Layers

Let us introduce the notation for individual LQN layers obtained by LN through loose layering. These are closed queueing networks with m identical c-server queues and a delay (i.e., infinite-server) node. Jobs in class r have service demand D_r at the multi-server station and think time Z_r at the infinite-server station.

Service distributions are assumed to be of phase type in LN. They include special cases such as the *Disabled* distribution, which allows users to forbid the routing of a class to a station for debugging purposes, and the *Immediate* distribution, which characterizes negligible processing that takes zero time. The handling of the latter is solver-dependent. For example, LINE's CTMC solver applies stochastic complementation to remove the corresponding transitions in an exact fashion [25].

We now outline the mapping between LQN abstractions and product-form models. Under loose layering, a layer l consists of a queueing network with m identical queueing stations, modeling servers in that layer, and a single infinite-server station, modeling the clients. Thus, m denotes the replication factor of the server. We shall refer to such a queueing network as a *homogeneous* model. The number of servers in the queueing stations is equal to the multiplicity c of the task or host processor acting as servers in that layer. Figure 2 gives the model of a layer for the LQN where T3 acts as server.

Clients issuing synchronous calls to layer l are represented as jobs initialized in a *reference class* at the delay. Subsequently, these jobs cycle between the delay and the queues, switching class to represent the specific tasks, entries, and activities that the clients visit (or invoke) during execution. Parameters such as think times and service demands are iteratively updated as per the method of layers [29]. For example, if certain sections of the client workflow require access to another layer $l' \neq l$, the corresponding residence times are modeled as think times that already incorporate the queueing contention in layer l'. Moreover, the probability of a client executing a particular entry is set proportional to the last throughput of this entry. Unlike LQNS, LN updates routing probabilities at each iteration, because not all solution paradigms are visit-based.

Clients that send asynchronous calls to layer l are instead represented as open Poisson arrival streams. Coexistence of open and closed classes therefore gives rise to mixed models, which can be reduced to closed ones by demand scaling [5].

Service classes are mapped to a set of R chains, obtained by computing the strongly connected components of the routing matrix. Each chain j represents a client task to the layer, and has an associated number of jobs N_j that are initialized at the infinite-server station, starting in the chain reference class. Note that this does not loses information as it is possible to exactly recover the per-class performance metrics from the per-chain ones [5,36].

For ease of presentation, since a multi-chain model can always be reduced to a corresponding multi-class model with R classes, one per chain, we shall use the terms "chain" and "class" interchangeably.

3.3 Performance Metrics

Performance metric computation is solver-dependent. We focus here on the default solver used by LN, which is LINE's normalizing constant (NC) solver. We assume for simplicity that scheduling leads to product-form models and single-server stations ($c = 1$). Approximations to handle other cases, such as multi-class FCFS, are discussed later in the paper.

Let $N = (N_1, \ldots, N_R)$ be the population vector for a layer, $|N| = \sum_{r=1}^{R} N_r$, and recall that $G(m, N)$ is the normalizing constant of the state probabilities for the associated product-form model, which consists of m identical queueing stations and an infinite-server node. Denote by 1_r a row vector of all zeros with a one in the r-th dimension. We may exploit the following relations for the mean class-r throughput $X_r(N)$ and for the mean class-r queue length $Q_r(N)$ at any of the identical queueing stations:

$$X_r(N) = \frac{G(m, N - 1_r)}{G(m, N)} \qquad Q_r(N) = D_r \frac{G(m+1, N - 1_r)}{G(m, N)}$$

The system throughput $X_r(N)$ is assumed to be computed at a reference station for which we set the mean number of visits of class r to unity. Little's law may then be combined with the previous relations to obtain other metrics such as mean response times and resource utilizations [14,22].

Before discussing the novel algorithms integrated in LN to compute these metrics, we remark that specific simplifications arise in evaluating normalizing constants for homogeneous layers due to the structure of the product-form solutions. At first, if either $Z_r = 0 \wedge D_r = 0$ or $N_r = 0$ holds for a class r, then this class can be removed from the model as it does not contribute to the normalizing constant. Define $\mathcal{R}_D$ as the set of remaining classes for which $D_r = 0$. We note that the contribution of such classes to $G(m, N)$ is given exactly by a factor $\prod_{s \in \mathcal{R}_D} Z_s^{N_s}/N_s!$. Hence, every model can be reduced without loss of generality to one where all classes have $D_r > 0$, which we will assume throughout.

3.4 Advanced LN Features

We now briefly overview advanced features and LQN extensions supported by LN, which are to the best of our knowledge unique to this solver.

Caching Layers. We have made an extension to the LQN formalism, enabling the inclusion of *cache* nodes. When visiting a cache, a job reads an item according to some probability, resulting in either a cache hit or a cache miss. The subsequent processing activities can depend on whether the read outcome was a hit or miss. Cache reads also activate a replacement policy (e.g., random replacement, FIFO, LRU) to evict infrequently used items. The LN solver features the ability to define caches in an LQN using specialized tasks and entries named CacheTask and ItemEntry, which capture the data access requirements of jobs traversing the LQN. More details can be found in [17] and in Sect. 5.

Multi-chain Joint and Marginal Probabilities. The LN solver allows the computation of joint and marginal state probabilities in each layer, leveraging the ability of the NC solver to evaluate normalizing constants. This makes it possible to obtain probabilistic measures, which may be useful for example in parameter inference and buffer overflow analysis.

Multi-chain Transient Analysis. With the FLUID solver, one can compute transient metrics and passage time distributions in each layer. This solver performs mean-field approximations for PS nodes based on the theory presented in [28]. As mentioned before, FCFS stations are also treated as PS nodes with service demands corrected through a hybrid $M/G/k$-diffusion approximation [9].

Response Time Distributions. Recently, we have demonstrated the possibility to couple LN with mixture density networks (MDNs) for response time distribution analysis [27]. The MDN-based approach considerably increases the precision of computing response time percentiles for LQNs compared to analytical approximations, which are notoriously difficult for multi-chain networks.

4 Novel Algorithms

LN's default layer solver, NC, implements state-of-the-art exact and approximate methods for normalizing constant analysis of mixed queueing networks. Historically, such methods were replaced in the early years of performance evaluation by exact and approximate MVA algorithms to overcome intrinsic numerical instabilities arising from the use of normalizing constants. However, recently developed techniques for computing normalizing constants exhibit superior complexity to their corresponding MVA counterparts, prompting a reconsideration of these methods, as we discuss throughout.

Particularly in the context of loose layering, we show that the normalizing constant numerical instabilities can be circumvented through appropriate scalings or log-sum-exp approximations [4], and propose several exact and asymptotic solution methods for queueing networks, that are not available in the traditional MVA framework.

4.1 Solving Homogeneous Layers with the CoMoM Algorithm

Recall that mixed queueing network models can be mapped with suitable transformation to a model consisting only of closed classes [5, §8.2.3]. On this basis, LN analyzes layers by default using the NC solver, which implements the Class-oriented Method of Moments (CoMoM) algorithm proposed in [6]. For a model with N jobs belonging to a fixed number of R classes, CoMoM implementations require approximately log-quadratic time and log-linear space in $|N|$ to obtain an *exact* solution, thus being more scalable than the exact MVA algorithms. The latter have a time and space complexity of $O(|N|^R)$. Moreover, contrary to other moment-based methods, CoMoM can avoid degeneracies when the model consists of one or more replicated queueing stations, as in the case of loose layering. An equivalent result is not currently available for MVA. Among the complications, it is worth noting that MVA expressions such as the celebrated arrival theorem are bi-linear in their defining terms, mean queue lengths and mean throughputs, yielding systems of non-linear equations that are not as tractable as CoMoM's linear matrix recurrence relation.

Enhancements. LN evolves CoMoM by developing explicit solutions to its system of linear equations for homogeneous models. Such solutions are applicable to models with an arbitrary number of classes R. For a vector v with d dimensions, let $|v| = \sum_{i=1}^{d} v_i$ and define $\mathrm{diag}(v)$ as a diagonal matrix with the elements of v placed on the main diagonal. We give the following exact result.

Theorem 1. *Consider a product-form queueing network model with R classes, having m identical single-server queueing stations with service demand $D_r > 0$ in class r and an infinite-server node with think time Z_r in class r. Define the collection of normalizing constants*

$$g(m, N) = \left[G(m, N)\ G(m, N - 1_1) \cdots G(m, N - 1_{R-1})\right].$$

and the basis

$$\Lambda(N) = \left[g(m + 1, N) \quad g(m, N)\right]^T$$

Then the following matrix recurrence relation holds

$$\Lambda(N) = (\boldsymbol{F}_{1,R} + N_R^{-1}\boldsymbol{F}_{2,R})\Lambda(N - 1_R) \tag{1}$$

in which

$$\boldsymbol{F}_{1,R} = \begin{bmatrix} D_R\boldsymbol{E}_{1,1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad \boldsymbol{F}_{2,R} = \begin{bmatrix} mD_R\boldsymbol{S} & Z_R\boldsymbol{S} \\ mD_R\boldsymbol{I} & Z_R\boldsymbol{I} \end{bmatrix}$$

$$\boldsymbol{S} = -m^{-1}\begin{bmatrix} -|\tilde{N}| - m & \tilde{Z}^T \\ -\mathrm{diag}(\tilde{D})^{-1}\tilde{N} & \mathrm{diag}(\tilde{D})^{-1}\,\mathrm{diag}(\tilde{Z}) \end{bmatrix}$$

where $\boldsymbol{E}_{1,1}$ is of order R with a single nonzero entry in position $(1,1)$, $\widetilde{N} = (N_1, \ldots, N_{R-1})^T$, $\widetilde{D} = (D_1, \ldots, D_{R-1})^T$, $\widetilde{Z} = (Z_1, \ldots, Z_{R-1})^T$, $\boldsymbol{I}$ is the identity matrix of order R, and $\mathbf{0}$ is the zero matrix of order R.

A proof of the theorem is in the Appendix A. Note in particular that the knowledge of $\Lambda(N)$ and $\Lambda(N - 1_R)$ is sufficient to determine the expression of all the mean performance metrics introduced in Sect. 3.3.

Termination conditions for the matrix recurrence relation (1) are obtained noting that $G(\cdot, 0) = 1$ and, whenever any element of N is negative, $G(\cdot, N) = 0$.

Table 1. Relative error and runtime upon computing $\log G(m, N)$ exactly.

Classes	*Total jobs*	*Method*	*Runtime [s]*
8	40	Convolution	0.0033
8	40	CoMoM (original)	0.0047
8	40	CoMoM (enhanced)	0.0014
8	400	Convolution	1.4201
8	400	CoMoM (original)	1.1433
8	400	CoMoM (enhanced)	0.0016
8	4000	Convolution	Memory exhausted
8	4000	CoMoM (original)	Timeout
8	4000	CoMoM (enhanced)	0.0017
8	10^6	Convolution	Memory exhausted
8	10^6	CoMoM (original)	Timeout
8	10^6	CoMoM (enhanced)	0.2591

Numerical Stabilization. In principle, to prevent floating-point range exceptions, the proposed solution can either be computed using exact or multi-precision arithmetic. In practice, we have observed that scaling at each step the vector $\Lambda(N)$ so that $|\Lambda(N)| = 1$, and removing the effect of such scaling only in the final result, is sufficient to sanitize numerical problems in practical uses, except for negligible numerical fluctuations. This makes the theoretical and implementation complexity identical and, empirically, much faster than using exact arithmetic.

To illustrate this, we consider a challenging model with $R = 8$ classes, where $m = 1$, $Z_r = r$ and $D_r = 10^{-r}$, $r = 1, \ldots, R$. Jobs are split equally across the classes. The exponential spacing of the demands and the large population make the analysis numerically challenging. We set a timeout of 10 s to solve a model. The original CoMoM leverages exact arithmetic in Java, whereas the enhanced method implements in MATLAB the recursion we have proposed in Theorem 1 using standard floating-point arithmetic. Table 1 shows numerical results, which corroborate the high scalability of the enhanced CoMoM for homogeneous models. Results are obtained on an AMD Ryzen 7 2700X Processor with 64 GB of RAM. Note that at population $|N| = 4000$ convolution becomes unviable due to excessive memory requirements, but since the normalizing constant reaches order 10^{-602} it would have anyway exceeded the floating-point range during execution. Scaling methods for Convolution have been proposed in [21], however it is not difficult in our experience to generate examples of large models where this technique still cannot prevent floating-point range exceptions. Instead, the enhanced CoMoM can also solve the largest model with 10^6 jobs, agreeing within the first 11 digits of $\log G(m, N)$ with the results of the logistic expansion (LE) proposed in [8], which is asymptotically exact. We have also observed in all cases that at least the first 6-digits of the mean per-class throughputs computed by the enhanced CoMoM were identical to the ones obtained by the AQL approximate MVA method [37]. As no other exact method can reach this model scale, it is difficult to rigorously verify exactness, yet the result suggests no, or at least negligible, presence of error accumulation.

4.2 Solving Homogeneous Layers with Gaussian Quadratures

As illustrated in the last numerical example, the CoMoM method has slightly increasing time requirements to analyze a single layer as the population grows. This also occurs as R increases, since the CoMoM basis has $2R$ elements. While tens or hundreds of milliseconds may be negligible for a single model, LQNs are solved iteratively and can feature many layers, hence solution times compound quickly. In large models, it is therefore useful to trade accuracy for speed. LN uses to this aim quadrature methods for integral forms of the normalizing constant.

A simple expression for the normalizing constant is given by the McKenna-Mitra integral form [24]. This is in general a multidimensional integral, with one dimension for each queueing station in the model. Thus, in a homogeneous model for a layer one would expect a m-dimensional integral. We show however that in homogeneous models this integral form takes the following simpler expression.

Theorem 2. *Under the same assumptions of Theorem 1, the normalizing constant of state probabilities for the queueing network model admits the following integral form*

$$G(m,N) = \frac{1}{(m-1)!\prod_{r=1}^{R} N_r!}\int_{u=0}^{+\infty} e^{-u}u^{m-1}\prod_{r=1}^{R}(Z_r + D_r u)^{N_r}du \qquad (2)$$

A proof is given in Appendix C. The main difficulty associated with evaluating $G(m,N)$ directly is that (2) is prone to numerical difficulties. This is because quadratures do not operate directly in the log domain and are therefore numerically sensitive to the magnitude of the factors under the integration sign, one being an exponentially decaying function (e^{-u}), the other being a polynomial of high order $|N|+m-1$. A novel strategy developed in the NC solver to evaluate (2) is to use Gaussian quadrature methods coupled with the log-sum-exp trick [4]. We have implemented both Gauss-Legendre and Gauss-Laguerre quadratures for (2), finding them empirically better suited at evaluating normalizing constants than MATLAB's default `integral` method and overall the best evaluation methods unless job populations are asymptotically large. We point to Appendix B for a brief introduction of both Gauss-Legendre and Gauss-Laguerre quadratures.

Generally, Gauss-Laguerre quadrature enables increasingly precise evaluations of (2) for growing values of its order K, however it also faces numerical difficulties for large number of jobs N, for which the quadrature weights and the integrand can display vastly different magnitudes. In such cases, we evaluate instead $\log G(m,N)$ in the quadrature summation by applying to the expression the log-sum-exp method, using in particular the implementation described in [4].

Numerical Example. We consider the same models considered for CoMoM and numerically evaluate the integral form (2). MATLAB's integral method is run with an absolute tolerance of 10^{-12}. Node and weights for the Gaussian quadratures are precomputed offline: due to numerical instability we can reach for the Gauss-Laguerre method up to order 300, while for Gauss-Legendre we could precompute weights up to order 20000 in the range $u \in [0, 10^6]$. We also include in the test the LE asymptotic expansion implemented in NC, which is a scalable method for models with few stations and many classes. The method applies a Laplace's approximation to the simplex integral form for the normalizing constant in [8]. Asymptotically the LE results are tight to the exact solutions.

Table 2. Relative error and runtime upon approximating $\log G(m, N)$.

Classes	*Total jobs*	*Method*	*Rel. error [%]*	*Runtime [s]*
8	40	MATLAB's integral	0.0000	0.0006
8	40	Gauss-Legendre	0.0000	0.0004
8	40	Gauss-Laguerre	0.0000	0.0010
8	40	Logistic expansion	-0.1249	0.0012
8	400	MATLAB's integral	0.0144	0.0005
8	400	Gauss-Legendre	-0.0001	0.0006
8	400	Gauss-Laguerre	-0.0001	0.0010
8	400	Logistic expansion	0.0033	0.0013
8	4000	MATLAB's integral	Unstable	0.0008
8	4000	Gauss-Legendre	-0.0006	0.0021
8	4000	Gauss-Laguerre	-0.0006	0.0010
8	4000	Logistic expansion	0.0003	0.0013
8	10^6	MATLAB's integral	Unstable	0.0008
8	10^6	Gauss-Legendre	-0.0592	0.0095
8	10^6	Gauss-Laguerre	0.2508	0.0011
8	10^6	Logistic expansion	0.0000	0.0013

The results are given in Table 2. Overall, we see that Gauss-Legendre is typically sufficient except in large asymptotic models where LE solutions are closer to optimal. The lower performance of Gauss-Laguerre is interpreted as being due to the restriction of using up to 300 nodes and weights in the interpolation due to numerical instability in their computation. Since Gauss-Legendre quadratures of order $K = 2n-1$ are exact for polynomials up to order n, and the normalizing constant is itself the integral of a polynomial of order $n = |N| + m - 1$, it is possible to use the $K = 2|N| + 2m - 3$ order as a threshold for when the quadrature will cease to be exact and switch afterwards to LE. For example, with Gaussian integration of order $n = 300$ and $m = 1$ the solution would switch to LE for $|N| \geq 6000$. Note that, on top of this, approximation errors are incurred in

Table 2 by the log-sum-exp trick used for numerical stabilization, which explains why small errors are incurred by Gauss-Legendre and Gauss-Laguerre also in cases where the quadrature should be exact. Another source of errors is that Gauss-Legendre requires a finite interval and has therefore been truncated to the range $u \in [0, 10^6]$, whereas the normalizing constant integral is defined in the range $u \in [0, \infty]$.

Summarizing, the numerical analysis reveals that a combination of Gauss-Legendre quadrature, for models with tens or hundreds of jobs, and LE, for larger models, provides an effective way to approximate homogeneous layers.

4.3 Computing Marginal Probabilities in Homogeneous Layers

Using normalizing constants instead of MVA simplifies the calculation of probabilistic measures on each layer, as illustrated in this section. Thanks to loose layering, specialized results can be derived to allow simple computation of marginal probabilities in a layer. We focus here in particular on the marginal probability $\pi_N(m, n)$ that n jobs are queueing or receiving services at any of the m identical queueing stations. This is also equal to the probability that $\pi_N(m, |N| - n)$ jobs are waiting at the infinite server station.

Computing $\pi_N(m, n)$ is in general a difficult problem, since with R classes there is a combinatorially-large number of job mixes that result in the same total job population n at the m queueing nodes. In this case, NC leverages a novel result, developed in the next theorem, which obtains marginal probabilities in $O(|N|^2)$ time and $O(|N|)$ space in homogeneous layers. For $m = 1$, this improves over MVA methods that require instead $O(|N|^R)$ time and space, while matching the complexity of CoMoM's extension to marginal probabilities [6, §VII], but without requiring the solution of a system of linear equations as CoMoM does for marginal probabilities. Another novelty is that, unlike CoMoM, the expression below applies also to homogeneous models with $m > 1$.

Theorem 3. *Under the same assumptions of Theorem 1, let $\pi_N(m, k)$ be the marginal probability that the m queueing stations have k resident jobs in total, $k = 0, \dots, |N|$. Define the following basis of unnormalized probabilities:*

$$\Pi(N) = G(m, N) \left[\pi_N(m, |N|), \dots, \pi_N(m, 0)\right]^T$$

with $\Pi(0) = (0, \dots, 0, 1)^T$. Then $G(m, N) = |\Pi(N)|$ and the following exact recurrence relation holds

$$\Pi(N) = N_R^{-1} \boldsymbol{T}_R \Pi(N - 1_R) \tag{3}$$

where

$$\boldsymbol{T}_R = \begin{bmatrix} Z_R & (|N| + m - 1)D_r & 0 & \cdots & 0 \\ 0 & Z_R & (|N| + m - 2)D_r & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & Z_R & (m - 1)D_r \\ 0 & 0 & 0 & 0 & Z_R \end{bmatrix}$$

A proof of the result is given in Appendix D. While the result is exact, this little says about its numerical stability. We have verified with numerical examples, using the load-dependent convolution algorithm, that the formulas in Theorem 3 match numerically the exact solutions, while avoiding exponential time and space requirements as the number of classes grows. In the tests we observe that the method is applicable using floating-point arithmetic only to models with up to, approximately, $|N| = 500$ jobs, provided that, without loss of generality [21], demands are rescaled beforehand to $D_r = 1$, $r = 1, \ldots, R$. Larger models require instead the use of exact or multi-precision arithmetic to prevent floating-point range exceptions, which heightens complexity by a log-linear factor in both time and space [7].

4.4 Multi-server Nodes, Load-Dependence and Multi-class FCFS

We here briefly discuss other strategies used in LN to accelerate the evaluation and cope with extended features. In cases where the scheduling policy does not yield a product-form, suitable approximations are coupled with the proposed algorithms to approximate the solution. In particular, first-come first-served (FCFS) multi-class stations are mapped to PS stations with demands iteratively adjusted with an interpolation that depends on a hybrid $M/G/k$-diffusion approximation, as proposed in [9]. We point to the original paper for results showing high accuracy.

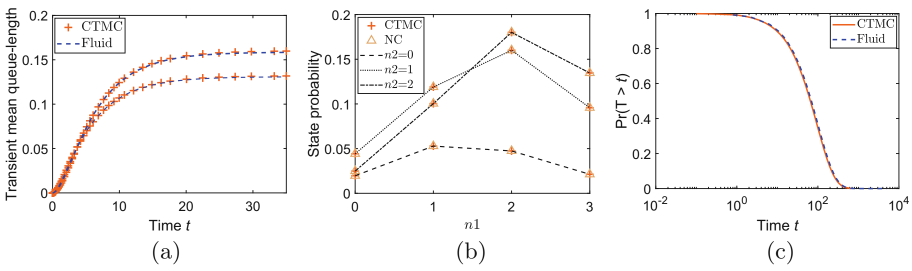


Fig. 3. Some meta-solver capabilities of LINE in analyzing LQN models.

Seidmann's approximation is used by default in NC to approximate FCFS and PS stations with multiple servers [31]. This is a simple method that replaces a c-server station with demands D_r with a sub-network consisting of a single-server queueing station having demands D_r/c and infinite server station having demands $D_r(c-1)/c$. Under this transformation, as in the original system, a class-r job can traverse the two stations in D_r time overall when these are found both empty upon arrival. Moreover, the new infinite server station delay can be exactly aggregated within the pre-existing infinite server think time, so that the model retains overall the same number of stations.

Load-dependent modeling methods are also available in LINE to evaluate individual layers, which rely on the exact normalizing constant methods recently proposed in [11]. In essence, the latter factorize the normalizing constant of a load-dependent model into solving a single-server queueing network and scaling the resultant mean performance metrics by the normalizing constant of a related load-dependent model defined on a reduced state space. The work shows that this can be done either exactly or approximately, based on mean-value analysis (RD method) or a novel integral form of the normalizing constant (Norlund-Rice form). Although these methods may also be applicable to multi-server station analysis, Seidmann's approximation often suffices to achieve good accuracy while retaining the benefit of reducing the problem to a simple single-server model on which the CoMoM and Gaussian integration methods both apply.

5 Case Studies

5.1 Meta-solver Capabilities

The most distinctive feature of the LN solver is the meta-solver capability. Different solution paradigms can be used throughout individual iterations and across layers. Moreover, once the iteration has reached a fixed-point, multiple paradigms can be applied to obtain the metrics of interest. We here focus on the latter case.

We illustrate this feature on the example shown in Fig. 1, which describes a scenario where two job classes T1 and T2 require services from a server T3. There are 3 and 2 jobs for *class*1 and *class*2 respectively, and T3 is a FCFS service station with multiplicity $c = 2$. We consider transient analysis, for which LINE provides multiple solver options. Figure 3a shows the transient average queue length for the two job classes, representing T1 and T2 as clients, for the layer where T3 is modeled as a server. The plot shows a tight matching between the figures given by both CTMC and FLUID solvers. We can observe that the system reaches steady-state at around $t = 16$. In the figure, FLUID solver looks very accurate for this non-saturated single queue scenario but the accuracy depends on load and number of servers.

We also show meta-solver capabilities on steady-state probabilities. Figure 3b displays the joint steady-state probabilities calculated by NC and CTMC solvers for the whole state space at T3, given the service demand of A3 as 50. The results from both solvers are almost the same, but the speed of NC solver is much faster. In this example, the calculation of each probability takes CTMC solver around 4 s while the NC solver takes less than 50 ms.

Beyond joint probabilities, LINE allows us to compute response time percentiles with the FLUID and CTMC solvers. In the model, the operations of T3 are executed by the processor P3, here we use both solvers the obtain the response time distribution at the layer where T3 acts as client to P3. Results are shown in Fig. 3c, demonstrating agreement of the solutions.

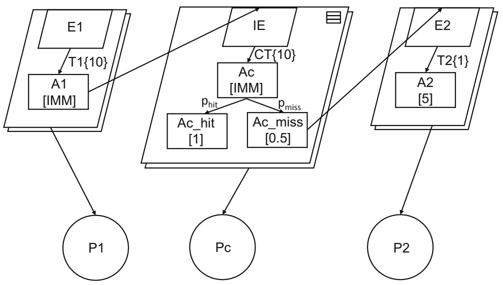


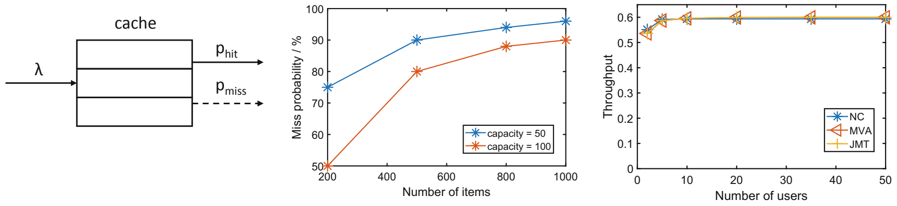
Fig. 4. Example of a multi-formalism model containing a caching layer (middle task)

5.2 Multi-formalism Capabilities

LINE can analyze, analytically or via simulation, models with integrated queueing and caching formalisms. LN can therefore also solve LQNs with caching, such as the three-layer model in Fig. 4. The host processors P1, Pc, P2 adopt the processor sharing scheduling policy. The number of users is represented by the multiplicity of the task T1 and the number of jobs is represented by the multiplicity of the task T2 and the cache task CT. As per Sect. 3.4, a cache task is a novel LQN element introduced by LN to describe data item reads from a cache, with different activities occurring based on whether a cache hit or a cache miss occurred. This is modeled by means of state-dependent class-switching.

In the example under study, items access probabilities obey a discrete uniform distribution and the cache is configured with a random replacement (RR) strategy. Arbitrary access probability distributions may be configured in LN and replacement strategies such as FIFO or LRU are supported. Jobs requested from T1 retrieve the items in the cache task CT. If the required items are cached, jobs will be processed by the *hit* activity with a probability of p_{hit}. Otherwise, jobs will be transformed to the *miss* activity with a probability of p_{miss} and be further processed by the task T2.

To solve an LQN model containing caching, LN first decomposes the entire model into a group of layers, as illustrated in Sect. 3. For the layer without cache nodes, the solutions are given in Sect. 4. On the other hand, for the layer



(a) Cache upper sub-model (b) Miss probability analysis (c) MVA and NC solutions

Fig. 5. Capabilities of LINE in analyzing multi-formalism LQN models

that involves a cache node, LN additionally divides the layer into two sub-models. In the upper sub-model, the cache node is isolated in an open model with Poisson arrivals, as shown in Fig. 5a. In the lower sub-model, the delay and the queueing station are contained in a closed queueing network with routing probabilities dynamically obtained from the p_{hit} and p_{miss} values obtained at the last iteration on the upper sub-model. More details can be found in [17].

For this model that combines both the queueing and caching stochastic formalisms, numerical results given by LINE are shown in Fig. 5. Figure 5b demonstrates the miss probabilities against different number of items, which decrease with the improvement of the cache capacity. Figure 5c compares the accuracy of the throughput for cache solved by MVA and NC solver respectively. MVA analyzes caches by the fixed-point iteration method (FPI) proposed in [10], whereas NC implements the normalizing constant asymptotics proposed in the same paper. The solvers are both compared against an equivalent model constructed with a JMT model using both queueing and Petri net formalisms, similar to the validation model used in [17], but adapted to the example at hand. The results indicate high accuracy of both MVA and NC in capturing the cache layer throughput, which is in general a function of the cache hit ratio.

6 Conclusion

We have presented the LN solver, the first meta-solver for LQNs, introducing new analysis methods for loose layering, in particular Gaussian integrals and a stabilized version of the exact CoMoM [6] to efficiently analyze layers in milliseconds. Case studies have shown the ability of the tool of combining several formalisms and solution methods in LQN analysis.

Future work will focus on extending LN to broaden the support for extended queueing models, such as fork-join networks and state-dependent queues.

Acknowledgments. 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).

A Proof of Theorem 1

For a homogeneous model, the CoMoM recurrence relation may be written as

$$\boldsymbol{A}\Lambda(N) = \boldsymbol{B}\Lambda(N - 1_R)$$

where

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} \\ \boldsymbol{0} & \boldsymbol{A}_{2,2} \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_{1,1} & \boldsymbol{0} \\ \boldsymbol{B}_{2,1} & \boldsymbol{B}_{2,2} \end{bmatrix}$$

Let $\boldsymbol{0}_{I,J}$ indicate a block of zeros of size $I \times J$. Defining $\tilde{D} = (D_1, \ldots, D_R)^T$, we have

$$\boldsymbol{A}_{1,1} = \begin{bmatrix} 1 & -\tilde{D}^T \\ \boldsymbol{0}_{R-1,1} & -m\,\mathrm{diag}(\tilde{D}) \end{bmatrix} \quad \boldsymbol{A}_{1,2} = \begin{bmatrix} -1 & \boldsymbol{0}_{1,R-1} \\ \tilde{N} & -\mathrm{diag}(\tilde{Z}) \end{bmatrix} \quad \boldsymbol{A}_{2,2} = N_R \boldsymbol{I}$$

$$\boldsymbol{B}_{1,1} = D_R \boldsymbol{E}_{1,1} \quad \boldsymbol{B}_{2,1} = m D_R \boldsymbol{I} \quad \boldsymbol{B}_{2,2} = Z_R \boldsymbol{I}$$

The inverse of the block upper triagonal matrix A is now computed as

$$\boldsymbol{A}^{-1} = \begin{bmatrix} \boldsymbol{A}_{1,1}^{-1} & \boldsymbol{S}\boldsymbol{A}_{2,2}^{-1} \\ \boldsymbol{0}_{R-1,1} & \boldsymbol{A}_{2,2}^{-1} \end{bmatrix}$$

with $\boldsymbol{S} = -\boldsymbol{A}_{1,1}^{-1}\boldsymbol{A}_{1,2}$. Observe first that

$$\boldsymbol{A}_{1,1}^{-1} = \begin{bmatrix} 1 & -m^{-1}e^T \\ \boldsymbol{0}_{R-1,1} & -m^{-1}\operatorname{diag}(\tilde{D})^{-1} \end{bmatrix} \quad \boldsymbol{A}_{2,2}^{-1} = N_R^{-1}\boldsymbol{I}$$

where $e^T = \tilde{D}^T \operatorname{diag}(\tilde{D})^{-1} = (1, \dots, 1)$. Thus

$$\begin{aligned} \boldsymbol{S} &= -\begin{bmatrix} 1 & -m^{-1}e^T \\ \boldsymbol{0}_{R-1,1} & -m^{-1}\operatorname{diag}(\tilde{D})^{-1} \end{bmatrix} \begin{bmatrix} -1 & \boldsymbol{0}_{1,R-1} \\ \tilde{N} & -\operatorname{diag}(\tilde{Z}) \end{bmatrix} \\ &= -m^{-1} \begin{bmatrix} -|\tilde{N}| - m & \tilde{Z}^T \\ -\operatorname{diag}(\tilde{D})^{-1}\tilde{N} & \operatorname{diag}(\tilde{D})^{-1}\operatorname{diag}(\tilde{Z}) \end{bmatrix} \end{aligned}$$

Note that $\boldsymbol{A}_{2,2}$ is the only block that depends on N_R. We can therefore write

$$\boldsymbol{A}^{-1}\boldsymbol{B} = \boldsymbol{F}_{1,R} + N_R^{-1}\boldsymbol{F}_{2,R}$$

where

$$\boldsymbol{F}_{1,R} = \begin{bmatrix} \boldsymbol{A}_{1,1}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} D_R\boldsymbol{E}_{1,1} & \boldsymbol{0} \\ mD_R\boldsymbol{I} & Z_R\boldsymbol{I} \end{bmatrix} = \begin{bmatrix} D_R\boldsymbol{E}_{1,1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}$$

$$\boldsymbol{F}_{2,R} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{S} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} D_R\boldsymbol{E}_{1,1} & \boldsymbol{0} \\ mD_R\boldsymbol{I} & Z_R\boldsymbol{I} \end{bmatrix} = \begin{bmatrix} mD_R\boldsymbol{S} & Z_R\boldsymbol{S} \\ mD_R\boldsymbol{I} & Z_R\boldsymbol{I} \end{bmatrix}$$

B Gaussian Quadratures

A Gauss-Laguerre quadrature of order K evaluates exponentially-weighted integrals by means of the approximation

$$\int_{x=0}^{\infty} e^{-x} f(x)dx \approx \sum_{k=1}^{K} w_k f(x_k) \tag{4}$$

where x_k denotes the k-th root of the Laguerre polynomial

$$L_K(x) = \sum_{j=0}^{K} \binom{K}{i} \frac{(-1)^j}{j!} x^j$$

and with weights $w_k = x_k \left((k+1)^2 \left[L_{k+1}(x_k) \right]^2 \right)^{-1}$.

Gauss-Legendre methods are similar but applicable to finite ranges $[a, b]$. Setting $a = 0$ and large b they can also help evaluating the normalizing constant. Their main benefit is that nodes and weights do not incur the same floating-point range exceptions as observed instead for Gauss-Laguerre quadratures of large order. We point to [20] for further details on Gauss-Legendre methods.

C Proof of Theorem 2

For a homogeneous model with m identical single-server stations, the McKenna-Mitra integral takes the form

$$G(m,N) = \frac{1}{\prod_{r=1}^{R} N_r!} \int_{u_1=0}^{+\infty} \cdots \int_{u_m=0}^{+\infty} e^{-(u_1+\ldots+u_m)} h(u_1+\ldots+u_m) du_1 \cdots du_m$$

where $h(u) = \prod_{r=1}^{R} (Z_r + D_r u)^{N_r}$. We note that the multidimensional integral may be interpreted as computing $E[h(U_1+\ldots+U_m)]$ for the i.i.d. exponential random variables $U_i \sim Exp(1)$. The result then readily follows after noting that $U_1+\ldots+U_m$ is Erlang-m distributed with density $f(u) = \frac{1}{(m-1)!} u^{m-1} e^{-u}$.

D Proof of Theorem 3

Let the entries of $\Pi(N)$ be indicated with $\widetilde{\pi}_N(n)$, $n = |N|, \ldots, 0$. A probabilistic population constraint holds for homogeneous models with $m = 1$ [6, Thm. 6]

$$N_R \widetilde{\pi}_N(n) = Z_R \widetilde{\pi}_{N-1_R}(n) + n D_R \widetilde{\pi}_{N-1_R}(n-1)$$

for all $n = 1, \ldots, |N|$ and where $\widetilde{\pi}_{N-1_R}(n-1) = 0$ if $n = 0$. With a load-dependent queueing station ($m = 1$), the derivation in [6] generalizes with similar passages to the following form

$$N_R \widetilde{\pi}_N(n) = Z_R \widetilde{\pi}_{N-1_R}(n) + n \frac{D_R}{\mu(n)} \widetilde{\pi}_{N-1_R}(n-1) \tag{5}$$

where $\mu(n)$ is the load-dependent scaling at the queueing station. Organizing (5) in matrix form, we get (3)

$$\boldsymbol{T}_R = \begin{bmatrix} Z_R & |N| \frac{D_R}{\mu(|N|)} & 0 & \cdots & 0 \\ 0 & Z_R & (|N|-1) \frac{D_R}{\mu(|N|-1)} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & Z_R & \frac{D_R}{\mu(1)} \\ 0 & 0 & 0 & 0 & Z_R \end{bmatrix}$$

As assumed, consider now an homogeneous layer, where there are m identical *load-independent* single-server stations. The proof follows by noting that, if $m > 1$, the m queueing stations can be exactly replaced by a flow-equivalent server station with identical $D_1, \ldots, D_R$ and [26]

$$\mu(n) = \frac{n}{(n+m-1)}$$

The final expression for $\boldsymbol{T}_R$ follows after plugging the above expression for $\mu(n)$.

E Software Architecture Design

Figure 6 illustrates the key architectural elements of LINE, including the NetworkStruct data structure, and the Network, NetworkSolver, LayeredNetwork, EnsembleSolver and LayeredNetworkSolver classes.

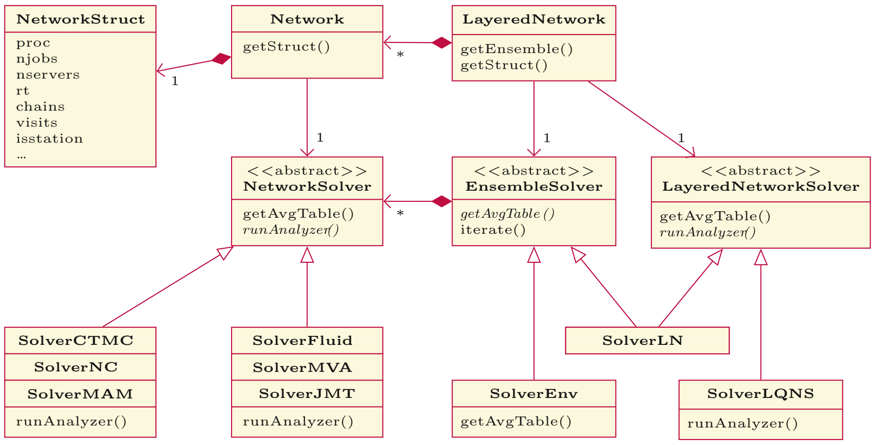


Fig. 6. Key architectural elements of LINE.

The Network object summarizes the model characteristics and acts as its persistence layer. The object is generated by the user either through a domain-specific language offered by LINE [9] or via model-to-model transformations from other formats (e.g., JMT's XML [2], PMIF [32]). Besides the model specification, a Network object can cache the model state space, its initial state, and retain information needed for the traffic equations in state-dependent models.

Each Network object is able to synthesize via the getStruct method a NetworkStruct data structure. The latter includes key model parameters, such as representations of service and arrival processes, job populations, and number of servers, among others. In addition, the data structure includes the routing table, the associated chains, and the average number of visits that each class pays to each node. NetworkStruct also offers indexing functions, that allow for example to differentiate between *stations*, where jobs can reside, and *nodes*, which are elements of the network traversed with zero service time (e.g., a fork).

The NetworkSolver object encodes a solver type, of which the aforementioned six LINE solvers are specific instances. The main role of this class is to ensure consistent computation of performance results, adopting identical conventions for reporting per-class and per-chain results to the end-user. Operational relationships are also applied by this object to derive certain performance metrics from the ones returned by the solvers, e.g. arrival rates from throughputs [14].

Each NetworkSolver object is equipped with a getAvgTable method that returns mean performance metrics for the model in a tabular format. The method invokes via the runAnalyzer method one of the solution methods offered by that solver, which operates solely on the NetworkStruct data structure. Model transformations that alter the model topology are conducted within runAnalyzer. An example is tagging a job class, which is used in response time distribution analysis.

The EnsembleSolver specifies the life-cycle for an iterative solution method that works on an ensemble of Network objects. This class allows to bind a Network-Solver to each particular Network in the ensemble, applying consistently actions before, in-between, and after each iteration, and verifying convergence. It also harmonizes the presentation of ensemble-level results to the end-user. Besides LN, the Env solver is another example of EnsembleSolver, wherein iteration is used to analyze random environments [13].

The LN solver is a special instance of EnsembleSolver, operating on an ensemble consisting of the LQN layers. The LayeredNetwork class encompasses the objects that form an LQN, such as the Entry, Task, Host, and Activity classes.

The LayeredNetwork class offers a getEnsemble method that generates, and stores within the LayeredNetwork object, the ensemble of Network models, each mapping to a distinct LQN layer. Similarly to Network, this class also exposes a getStruct method that builds a static data structure of the LQN parameters.

The LN solver, implemented in the SolverLN class, is specified parametrically in terms of any of the LINE solvers, or a custom combination thereof. For example, the end-user may require to use LINE's simulators on layers that include non-Markovian service distributions (e.g., Pareto) and MVA otherwise.

References

1. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 Years of GreatSPN. In: Fiondella, L., Puliafito, A. (eds.) Principles of Performance and Reliability Modeling and Evaluation. SSRE, pp. 227–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30599-8_9
2. Bertoli, M., Casale, G., Serazzi, G.: The JMT simulator for performance evaluation of non-product-form queueing networks. In: Proceedings of ANSS, pp. 3–10. IEEE (2007)
3. Bini, D., Meini, B., Steffé, S., Pérez, J.F., Van Houdt, B.: SMCSolver and Q-MAM: tools for matrix-analytic methods. ACM SIGMETRICS Perform. Eval. Rev. **39**(4), 46–46 (2012)
4. Blanchard, P., Higham, D.J., Higham, N.J.: Accurately computing the log-sum-exp and softmax functions. IMA J. Numer. Anal. **41**(4), 2311–2330 (2021)
5. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. John Wiley & Sons, Hoboken (2006)
6. Casale, G.: CoMoM: efficient class-oriented evaluation of multiclass performance models. IEEE Trans. Softw. Eng. **35**(2), 162–177 (2009)
7. Casale, G.: Exact analysis of performance models by the method of moments. Perform. Eval. **68**(6), 487–506 (2011)
8. Casale, G.: Accelerating performance inference over closed systems by asymptotic methods. In: Proceedings of ACM SIGMETRICS, pp. 8:1–8:25. ACM (2017)

9. Casale, G.: Integrated performance evaluation of extended queueing network models with LINE. In: Proceedings of WSC, pp. 2377–2388. IEEE (2020)
10. Casale, G., Gast, N.: Performance analysis methods for list-based caches with non-uniform access. IEEE/ACM Trans. Netw. **29**(2), 651–664 (2021)
11. Casale, G., Harrison, P.G., Ong, W.H.: Facilitating load-dependent queueing analysis through factorization. Perform. Eval. **152**, 102241 (2021)
12. Casale, G., Muntz, R.R., Serazzi, G.: Geometric bounds: a noniterative analysis technique for closed queueing networks. IEEE Trans. Comput. **57**(6), 780–794 (2008)
13. Casale, G., Tribastone, M., Harrison, P.G.: Blending randomness in closed queueing network models. Perform. Eval. **82**, 15–38 (2014)
14. Denning, P.J., Buzen, J.P.: The operational analysis of queueing network models. ACM Comput. Surv. **10**(3), 225–261 (1978)
15. Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S.: Enhanced modeling and solution of layered queueing networks. IEEE Trans. Softw. Eng. **35**(2), 148–161 (2009)
16. Franks, R.G.: Performance Analysis of Distributed Server Systems. Ph.D. thesis, Carleton University (2000)
17. Gao, Y., Casale, G.: JCSP: Joint caching and service placement for edge computing systems. In: Proceedings of IEEE/ACM IWQoS. IEEE (2022)
18. Gunther, N.J.: Analyzing Computer System Performance with Perl::PDQ. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22583-3
19. Horváth, G., Telek, M.: BuTools 2: a rich toolbox for Markovian performance evaluation. In: Proceedings of VALUETOOLS, pp. 137–142. ICST (2017)
20. Johansson, F., Mezzarobba, M.: Fast and rigorous arbitrary-precision computation of Gauss-Legendre quadrature nodes and weights. SIAM J. Sci. Comput. **40**(6), C726–C747 (2018)
21. Lam, S.S.: Dynamic scaling and growth behavior of queueing network normalization constants. J. ACM **29**(2), 492–513 (1982)
22. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice Hall, Hoboken (1984)
23. Marzolla, M.: The octave queueing package. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 174–177. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10696-0_14
24. McKenna, J., Mitra, D.: Asymptotic expansions and integral representations of moments of queue lengths in closed Markovian networks. J. ACM **31**(2), 346–360 (1984)
25. Meyer, C.D.: Stochastic complementation, uncoupling Markov chains, and the theory of nearly reducible systems. SIAM Rev. **31**(2), 240–272 (1989)
26. Mitra, D., McKenna, J.: Asymptotic expansions for closed Markovian networks with state-dependent service rates. J. ACM **33**(3), 568–592 (1986)
27. Niu, Z., Casale, G.: A mixture density network approach to predicting response times in layered systems. In: Proceedings of MASCOTS, pp. 1–8. IEEE (2021)
28. Pérez, J.F., Casale, G.: LINE: evaluating software applications in unreliable environments. IEEE Trans. Reliab. **66**(3), 837–853 (2017)
29. Rolia, J.A., Sevcik, K.C.: The method of layers. IEEE Trans. Softw. Eng. **21**(8), 689–700 (1995)
30. Ruuskanen, J., Berner, T., Årzén, K.E., Cervin, A.: Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. Perform. Eval. **151**, 102231 (2021)

31. Seidmann, A., Schweitzer, P.J., Shalev-Oren, S.: Computerized closed queueing network models of flexible manufacturing systems: a comparative evaluation. Large Scale Syst. **12**, 91–107 (1987)
32. Smith, C.U., Lladó, C.M., Puigjaner, R.: Performance model interchange format (PMIF 2): a comprehensive approach to queueing network model interoperability. Perform. Eval. **67**(7), 548–568 (2010)
33. Tribastone, M.: A fluid model for layered queueing networks. IEEE Trans. Softw. Eng. **39**(6), 744–756 (2013)
34. Waizmann, T., Tribastone, M.: DiffLQN: differential equation analysis of layered queuing networks. In: Companion of ICPE, pp. 63–68. ACM (2016)
35. Woodside, M.: Tutorial introduction to layered modeling of software performance (2013). http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf Accessed 08 May 2022
36. Zahorjan, J.: An exact solution method for the general class of closed separable queueing networks, vol. 8, pp. 107–112, New York, NY, USA (1979)
37. Zahorjan, J., Eager, D.L., Sweillam, H.M.: Accuracy, speed, and convergence of approximate mean value analysis. Perform. Eval. **8**(4), 255–270 (1988)
38. Zhu, L., Casale, G., Perez, I.: Fluid approximation of closed queueing networks with discriminatory processor sharing. Perform. Eval. **139**, 102094 (2020)