Simulating Layered Queueing Networks with Passive Resources

Greg Franks
Department of Systems and Computer Engineering
Carleton University
Ottawa ON K1S 5B6
greg@sce.carleton.ca

Keywords: Layered Queueing Network Simulation, MARTE UML Profile, Passive Resources, Logical Resources

Abstract
This paper describes an extension to Layered Queueing Networks (LQN), a form of an extended queueing network used to investigate performance problems, to model passive resources such as counting semaphores and buffers. Layered queueing networks can be constructed directly, or from UML design models which incorporate the MARTE profile, either directly or via the Core Scenario Model. Layered Queueing Networks scale well and can solve analytically systems with nested resource requests to active resources. However, passive resources cause problems which force the use of simulation. The layered queueing network simulator, lqsim, is also described here. Simulations are created by reading in an LQN model, constructing objects from pre-existing templates, then solving. The semaphore task extension was incorporated by modifying the existing template used to model multi-server tasks. Finally, the semaphore extension was used to solve a model of a building security system which has a pool of buffers to capture video images. The results here show that a lack of buffers is indeed a bottleneck, but other parts of the system ultimately limit the capacity of the system.

1. INTRODUCTION
There is now a large body of research into model-based performance prediction which attempts to integrate performance analysis into the software development process [1]. This approach incorporates annotations for performance analysis into a design model, then translates the design model into a performance model which is then solved using traditional approaches such as queueing networks, Petri nets or simulation. In the general case, there are $N$ “different” design models (not withstanding the various versions of UML, its design tools and the performance profiles), and $M$ different solution techniques. This variety of models and solution technique implies that there can be up to $N \times M$ different model converters.

The best approach to handle this form of problem is to use a common “intermediate” model between the $N$ design models and the $M$ performance models, thereby reducing the scope of the problem to $N + M$. This is the idea behind the Core Scenario Model (CSM) [8]. It’s meta-model was designed to closely match the domain model of the UML Profile for Schedulability, Performance and Time (SPT) [6], but only retains the information necessary for building a performance model. It can also be used with non-UML software specifications for input, and has produced as output regular and layered queueing networks, and stochastic Petri Nets [12].

Each performance model solution technique has its strengths and limitations. Those based on Markov Models such as stochastic Petri nets suffer from state space explosion. Regular queueing networks, especially when solved using approximate Mean Value Analysis, scale well but are not suited to solve models with nested resource requests, often found in modern multi-tier client-server software. Layered queueing networks can solve analytically models with nested active resource requests, but cannot handle resource requests to passive resources such as semaphores, locks and buffers when the acquire and release steps are performed by different tasks. The work in this paper describes an extension to layered queueing networks to overcome this last obstacle. The model is solved using simulation at this time.

This paper describes enhancements to the layered queueing network model and its simulator, lqsim, to solve performance models with nested requests to passive resources. Section 2 describes the core scenario model. Section 3 describes the Layered Queueing Network model and the semaphore task extension, used to model counting semaphores and buffers. Section 4 describes the layered queueing network simulator. Section 5 presents a performance model of a building security system, described previously in [13][12], and [7] §17.4.3]. Finally, section 6 concludes the paper.

2. CORE SCENARIO MODEL (CSM)
The Core Scenario Model [8] is an intermediate language used to describe performance formalisms which can then be easily translated into a final performance model. It gathers information found in UML deployment and behaviour models and combines this data with performance annotations from the UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) [7] into a form suitable for generating a performance model. In doing so, it also ensures the completeness of the information supplied, and supplies reasonable default values where possible otherwise. It is also possible to trace the elements of a CSM back to the UML model.
A CSM consists of one or more Scenarios which utilize various types of Resources. Figure 1 shows the major components of the Core Scenario Model (subclasses of PathConnection and Workload have been omitted). They are:

1. **Workload**: the intensity at which a scenario is executed within a system. Workloads can be either closed, with a fixed number of customers, or open, with a specified arrival rate.

2. **Scenario**: a set of steps used to accomplish some functional requirement in the system. Scenarios are composed of:
   - **Step**: a basic unit of work in a scenario which executes on the processor identified by its host processing resource. A CSM step corresponds to UML objects stereotyped with «PaStep».
   - **Path Connection**: used to connect steps. Sequences, or branches and merges, and forks and joins, and loops are all supported.

3. **General Resource**: used to indicate some object of limited capacity which must be acquired by a step in order for the step to perform its function. Resources have queues with some sort of discipline such as First-Come, First-Served. Resources are either passive or active.
   - **Passive Resource**: a resource which does not consume "time" on its own. Rather, it must also utilize an active resource (a host) to accomplish work. Logical resources, stereotyped as «GaLogicalResource», such as semaphores, buffers and locks, do not require a host. Resource requests and releases to logical resources are stereotyped as «GaAcqStep» and «GaRelStep» respectively.
   - **Component**: a passive resources which requires a host. This is typically a process, task or thread and is stereotyped as «PaRunInstance».
   - **Active Resource**: a node which consumes time such as a processor or a disk.
   - **Processing Resource**: an active resource that is scoped within the current model. Objects of this type are stereotyped as «GaExecHost».
   - **External Operation**: any active resource that is not contained within the model (for example, a remote server).

It is not strictly necessary to go through the CSM to produce a layered queueing network. They can be constructed directly from UML design models, for example [9][3]. However, experience has shown that various UML tools store internal data differently and that major revisions to UML itself force changes to the conversion software. It is hoped that using the CSM can minimize these difficulties.

![Diagram of Core Scenario Model](image)

**Figure 1.** Major components of the Core Scenario Model (from [3]).

### 3. LAYERED QUEUEING NETWORKS

A Layered Queueing Network (LQN) model [11][10][2] is a type of extended queueing network particularly well suited for modeling systems consisting of clients making requests to servers. The processors, processes or threads, and remote procedure calls that the threads make to each other in the system being studied are all represented directly in the LQN model. Layering arises because clients in the system make requests to servers who in turn make requests to other servers. Typically, these servers are all "active", in that they are either tasks running on processors, or devices such as CPU’s and disks. This work extends the LQN model by adding the capability of modeling passive resources (specifically, counting semaphores).

Figure 2 shows the major components of the meta model used to describe a layered queueing network.

- **Processors** are used to execute activities. They consume time and are always pure servers. Processors can either schedule tasks directly, or can schedule tasks in groups.
- **Groups** are used to collect a set of tasks together for the purpose of “completely fair scheduling” [4].
- **Tasks** are used to model customers in the model, software entities such as processes, passive resources such as semaphores, and non-processor devices such as disks. Requests are queued at tasks and are serviced by entries. Objects designated as Components in the CSM are modeled as tasks. Similarly, all External Operations are modeled as tasks, except that these tasks run on their...
own processors. Finally, all Closed Workloads are modeled as reference tasks that also run on their own processors. They typically have (but are not restricted to) one entry and these entries never accept requests.

Tasks are shown using large parallelograms in the LQN models shown here. A stacked task icon denotes multiple customers for reference tasks, and multi-servers otherwise.

- **Entries** are used to differentiate services provided by a task. They invoke activities. They are represented using parallelograms within task icons.

- **Activities** are the lowest level of modeling detail. From one to three activities can be connected implicitly in a sequence, called phases, or they can be connected into an execution graph explicitly through precedence elements. Steps in the CSM are modeled as activities.

- **Precedence** are used to connect activities. Tasks can spawn heterogeneous threads by using activities connected through AND-forks. OR-forks and activity looping are also supported. Path Connections in the CSM that do not cause a change of the Active Resource are modeled as precedence connections. Otherwise, Path Connections are modeled as Requests in a LQN model.

- **Requests** are used to send messages from activities in one task to entries on another. Requests can be synchronous, where the sender is blocked until the receiver replies, or asynchronous, where the sender and receiver proceed in parallel. Replies can be forwarded to another task which may reply to the original sender, or which may forward the request to yet another task. Synchronous messages are shown in the figures below using solid arcs with closed arrow heads, asynchronous messages are shown using solid arcs with open arrow heads, and forwarded messages are shown using a dashed line. Path Connections which cause a scenario to change from one Active Host to another are modeled as requests.

### 3.1. Non-Layered Queueing Networks

The Layered Queueing Network Model was developed to handle nested requests which works quite well for active and passive resources which process incoming requests to completion, for example remote procedure calls. The assumption is that tasks can service requests from any entry once service at an entry has completed (effectively, the serving task is stateless). Passive resources such as semaphores, locks and buffers violate this assumption. This section describes an extension to the LQN model to accommodate this type of behaviour, in particular, for counting semaphores. For the purposes of modeling, counting semaphores can be used for buffers too. Read-write locks, which are not covered here, can be accommodated easily by this technique.

Semaphores have been added to the LQN model by extending the existing Task-type to include a SemaphoreTask. Semaphore tasks behave like ordinary tasks except that they must have exactly two entries, one for wait and one for signal, and that each entry has its own queue for incoming requests. These entries are just like regular entries in that they can consume CPU time and can send requests to other entries. To model a purely passive resource, these entries should have zero service demand and make no requests to other servers. Semaphore tasks can be initialized as being either full or empty.

Signaling and waiting at the semaphore task is accomplished through requests. Wait operations must always be performed using synchronous requests because the caller must be blocked if the resource is unavailable. Signals can be performed using either synchronous or asynchronous requests. Requests to a semaphore task’s signal and wait entry must be made from separate activities because the order of requests originating from an activity is non-deterministic. Two approaches can be used:

1. The signal and wait can be generated from separate tasks, shown using forwarding in Figure 3a. If the signal follows the wait (a mutex), then all connections between the wait and signal entries must be deterministic.

2. Separate activities can be used to call the signal and wait. In Figure 3b, this relationship is shown using phases; the wait occurs in phase 1 and the signal in phase 2.
Figure 3. Methods of modeling non-nested counting semaphore resources using a LQN.

4. LAYERED QUEUEING SIMULATOR

This section describes the layered queueing network simulator, lqsim. Lqsim uses pre-defined templates that correspond to the elements shown in Figure 2 to construct a simulation corresponding to the input model. These templates are implemented using the underlying modeling elements of Parasol [5], described next. The details of the templates follow.

4.1. Parasol

Parasol [5] is an event-driven discrete simulator which is used to both simulate large-scale distributed systems for answering performance-related questions, and an emulator capable of emulating the operation of a concurrent system at the code level. It was designed to be efficient in terms of simulation speed and portable to a wide range of hardware and software platforms including Windows and Linux. To simplify the construction of simulation models of distributed systems, Parasol provides a virtual machine which supplies building blocks such as processors, busses, and hardware locks. Figure 4 shows the major elements of Parasol (those used for the Layered Queueing Network Simulator). They are:

- **Nodes** are collections of one or more hosts, used to model either a single CPU or are tightly-coupled homogeneous multiprocessor serving a common ready-to-run queue. Hosts are connected to each other through busses and links.

- **Busses** are shared communication channels used to connect two or more nodes to each other. Communication is bi-directional.

- **Links** are used to connect one node to another through a uni-directional communication channel.

- **Groups** are used to collect tasks together for completely fair scheduling [4].

- **Hosts** (or processors) are used to run tasks.

- **Tasks** are used to model everything from the threads in the system being modeled, to variants of ports, hosts, busses and links. They communicate to other tasks, optionally over busses and links, by sending asynchronous messages to ports.

- **Ports** are used to queue messages for reception by tasks. All tasks have a pre-defined std_port which is known by all other tasks. Tasks may also have other ports; these additional ports can be shared and can be transferred to other tasks.

Figure 5 shows the major states of Parasol used by Lqsim. User-defined code is executed by the simulator whenever the corresponding simulator task is in the Hot state. This code can suspend itself through ps_sleep(), send a message to a port through ps_send(), accept a message from a port through ps_receive(), or “consume time” on a host through ps_compute(). Note that several tasks can be in the Computing state, but only one task can be Hot. Tasks switching to and from the Hot state always cause a context switch.

4.2. Lqsim Simulation Construction

Parasol tasks are the primary object used to create a layered queueing simulation and are used to model all of the
task types found in an LQN model (Parasol nodes are used to model LQN processors). Listing 1 shows the main server loop used as a template for all LQN task types. This procedure is invoked upon the reception of a message from another task, or periodically for reference tasks (customers).

Entries in the LQN model are implemented through message types, shown as the entry attribute to the input message in lines 5 and 6 in Listing 1. This attribute is used to set the service time for this message and the potential list of destinations for requests. Tasks which are running on infinite-server processors call ps_sleep() at line 5; ps_compute() is used otherwise.

The LQN model supports three types of requests: synchronous, asynchronous, and forwarded. The server_cycle() loop is used for all three. For sending synchronous requests, the requesting task sets the reply_port attribute of the outgoing message at line 10, then waits for a response at line 12. (Messages sent to the task’s std_port will queue.) If asynchronous requests are being sent, then line 12 is omitted. Finally, if a synchronous request is being forwarded, the reply_port from msg_in is used to set the reply_port in msg_out. Line 12 is omitted, just as with the asynchronous message case. After this entry’s processing is completed, an optional reply is sent at line 17. This step is omitted for forwarded and asynchronous requests.

The subsections that follow describe the main task loops for tasks used to implement customers (reference tasks), single-threaded tasks, tasks with homogeneous threads (multi-servers), and semaphore tasks.

4.2.1. Reference and Single-threaded Tasks
Listing 2 shows the main task loop for single-threaded, non-semaphore tasks. These tasks are implemented using a single Parasol task with two ports. Messages sent from other tasks (requests) are delivered to the task’s pre-defined std_port. The second port, reply_port, allocated during task initialization, are used for replies to requests originating from this task’s server_cycle(). For reference tasks (customers), the receive() at line 3 is omitted, and a nil message is passed to server_cycle().

4.2.2. Multi-server Tasks
Parasol does not directly support multi-server tasks. Rather, a queue task running on its own virtual processor is used to dispatch requests to a set of worker tasks. Listing 3 shows the main task loop for the queue task. Workers queue up on the queue task’s worker_port waiting for requests from the std_port. If no workers are available, the queue task will block until one becomes available. Once a worker is available, the request received at line 3 is forwarded to the worker at line 6. This template is also used to implement infinite-server tasks. If no workers are queued at the receive() at line 4, then a new worker is created. The queue task itself consumes no “time” and runs on its own processor so that it does not block because of processor contention.
Worker tasks are based on the template in Listing 2 except that a send() to the worker’s queue task is inserted before the receive() at line 3, to signal the worker’s availability.

### 4.2.3. Semaphore Tasks

Semaphores are implemented using three task types, shown in Figure 6. The queue task, shown earlier in Listing 3, is used to handle semaphore wait requests. The signal task, used to handle semaphore signal requests is also implemented as a queue task. Neither queue task consumes “time” and they both run on their own virtual processor. Finally, a set of specialized token tasks do the actual semaphore processing (a counting semaphore is implemented by using more than one token task). The token task handles both signal and wait processing and can be purely passive, where it too consumes no time and runs on its own virtual processor, or active like other serving tasks. Note that a single input queue typically handles all entries for LQN tasks. However, for counting semaphores, wait and signals must be handled independently, so separate queues are implemented using independent Parasol tasks.

Listing 4 shows the code for the semaphore token task. This code is similar to the code used for a multi-servers’s worker in that it sends a “ready” messages to the semaphore’s wait and signal dispatcher. These messages are sent sequentially so that a signal always follows a wait. Further, the ready message sent to the signal task is sent as soon as the incoming message containing the wait entry payload arrives (at line 5) so that the semaphore’s signal task is ready to send a signal. This causes queueing to occur at the semaphore task rather than at the signal task so that asynchronous signal messages such as the one shown in Figure 6 are never delayed. If blocking behaviour for a signal is required, the task sending the signal request can use a synchronous request, thus being forced to wait for a reply from the second server_cycle() at line 8. This code models a lock, where the initial value of the semaphore is the same as the number of tokens. For the case where the initial value of the semaphore is zero, the order of the sends and their corresponding server cycles in Listing 4 is reversed.

### 5. BUILDING SECURITY SYSTEM MODEL

This section describes a model of a building security system, originally described in [13], where the performance impact of the number of buffers used to store video images is investigated. This system consists of a set of N security cameras for video surveillance, and a set of card readers and door locks used for access control. Only the video surveillance system is considered here.

Figure 7 shows the interaction diagram for the Video Acquire/Release scenario (from [12, 13]).

![Translated model for lqsim](image)

In Figure 8, the LQN model derived from the interaction diagram was then converted to the LQN model shown in Figure 8 by way of the CSM (described in [12]). For this work, the buffer task was converted to a semaphore task.

![Translated model for lqsim](image)
BufferManager \text{(ST)} and Database \text{(10)} are both active servers (a task which consumes CPU time), and a passive resource. Entries BM1 and BM2 correspond to allocBuf and releaseBuf in Figure 7 and perform the semaphore wait and signal operations respectively.

Figure 9 shows the results from the model varying the number of buffers, \( T \), from 1 to 12, and using \( N = \{30, 50, 70\} \) cameras. With \( N = 30 \) cameras, slightly more than one buffer is sufficient. The utilization at task AcquireProc is always less than one, and asynchronous messages from the cameras are never lost. However, to minimize the utilization AcquireProc, Figure 9 shows that four buffers are necessary. With \( N = 50 \) cameras, the system needs a minimum of five buffers. With less than five buffers, messages are lost at AcquireProc because the task is fully utilized due to blocking at the semaphore. With \( N = 70 \) cameras, no amount of buffers can prevent message loss: the system has a software bottleneck at AcquireProc, but this time it is due to insufficient resources elsewhere in the system. In all cases, the processor is never fully utilized.

6. CONCLUSIONS

This paper has presented an extension to the Layered Queueing Network model which allows modeling of passive resources such as buffers and counting semaphores through the use of a new task type called a Semaphore Task. These performance models can be created from a UML design model with appropriate annotations from the MARTE UML profile, or they can be constructed directly from a UML model, or can be constructed by hand.

The semaphore task introduced here behaves like other serving tasks already in the LQN model, except that it must have exactly two entries, one for signal and one for wait, and each of the two entries has its own queue. Strictly logical resources can be modeled easily by specifying entry service demands of zero time units. Conversely, the signal and wait entries can call other entries in the model, and they can consume time on a host processor. Boolean semaphores are modeled using a fixed-rate semaphore task; counting semaphores are modeled using a task multi-server.

The existing layered queueing network simulator, Lqsim, built upon the Parasol simulation framework, has been extended to support the semaphore task type. LQN simulation models are built from pre-existing templates constructed from Parasol objects. The semaphore task was constructed as an extension of the pre-existing multi-server task. The advantage of the approach taken here is that simulations can be constructed as quickly as the LQN input model parser can read and process an input file.

Finally, the Building Security System model, which appears in Section 17.4.3 of [2], can now be modeled taking buffers into account. The results here clearly show that the system needs more than one buffer to execute at capacity. Previous modeling efforts by our group with this model has ignored this constraint.

In the future, the semaphore task described here will be extended to handle read-write locks. A multi-entry, multi-queue task will be necessary with separate queues for the read acquire, write acquire and release operations.

REFERENCES


Figure 9. Results (with confidence intervals) from simulating the model in Figure 8.


