Tutorial Introduction to Layered Modeling of Software Performance

Murray Woodside Greg Franks

Department of Systems and Computer Engineering
Carleton University
Ottawa ON K1S 5B6
{cmw,greg}@sce.carleton.ca

January 31, 2023

Contents

1 Introduction

This note introduces the conceptual basis of layered queueing networks (LQNs) as a performance model for software systems, and then leads the reader through the features provided by the LQML modeling language. Layered queueing is an elegant compact notation for Extended Queueing Networks, which incorporates a number of features which are common in software systems (and other kinds of systems too). The central feature is a kind of structured "simultaneous resource possession", common in layered and client-server architectures, which gives it the name Layered Queueing.

There is also a User Manual for the LQNS solver and LQSIM simulator, and a thesis [**?**] and more recent technical paper [**?**] which summarize the underlying mathematics; these and other materials can be found on the layeredqueues.org web site, and on the LQNS page at `www.sce.carleton.ca/rads/lqns`.

2 Concepts

Performance, in the sense of response times and throughputs, is determined by how the system's operations use its resources. While a resource is in use it is busy, and other requests for it must wait, as determined by a queueing discipline. Ordinary queueing network (QN) performance models are directly applicable to systems in which each operation requires only a single resource, such as a CPU or disk; extended queueing networks (EQNs) are needed for simultaneous resources.

Software systems often have software servers that also have a queue of requests. When a request is accepted by the software server process, the process must then request and wait for the CPU. When it has the CPU, the operation can be executed and a reply returned for the request. This is "simultaneous resource possession" with two layers of resources, the software server and the CPU. The service time of the CPU is the host demand of the operation; the service time of the task includes the waiting for the CPU as well, possibly waiting for several "slices" of CPU time.

We consider layered systems (software systems, and other kinds of systems too) that are made up of servers (and other resources which we will model as servers); the generic term we will use for these entities is "task". A server is either a pure server, which executes operations on command (for instance a processor), or a more complex logical bundle of operations, which include the use of lower layer services. Such a bundle of operations may be implemented as a software server.

Figure 1 illustrates the elements of a software server, as they might be implemented in a software process. The threads are servers to the queue, and the requests take the form of interprocess messages (remote procedure calls, or the semantic equivalent), and the entries describe or define the classes of service which can be given. The assumption in this theory is that each thread has the capability of executing any entry. The execution of the entry can follow any sequence of operations, and can include any number of nested requests or calls to other servers. Calls to internal services of the server are assumed to be included in the entry, so all calls are to other servers. A canonical sequence of operations is the first-phase/second phase sequence shown by the heavy line in the figure, with the reply sent after the first phase. Software servers often send the reply as early as possible, and complete some operations later (e.g. database commits).

The execution of the server entity is assumed to be carried out by a host processor, which may be a multiprocessor (not shown in Figure 1). Once the request is accepted, the execution of the entry is a sequence of requests for service to the host and to other servers, and the essence of layered modeling is to capture this nesting of requests. Each request requires queueing at the other server, and then execution of a service there. The service time of a software server is the time to execute one request, and it has two components, a phase 1 time until the reply is generated, and a phase 2 time.

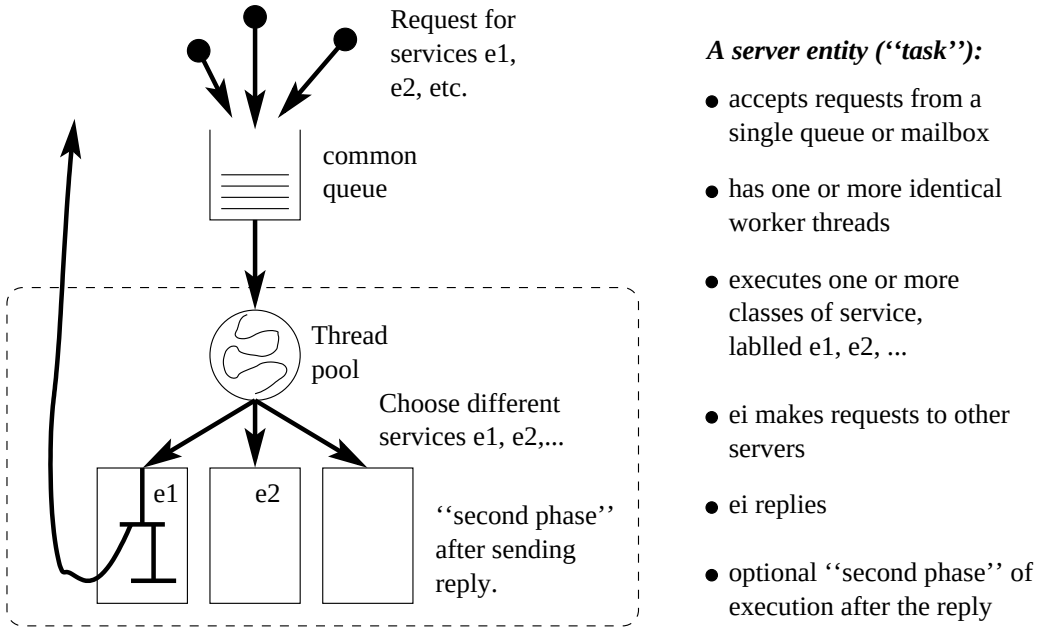


Figure 1: The elements of a software server, including multiple threads, multiple entries, and second phases, which are all optional.

The "thread" abstraction in a software server can also model other mechanisms that allow the server to operate simultaneously on several requests, including a process pool, kernel or user threads, dynamically created threads or processes, and virtual threads.

The "task" concept created for a software server can also be applied to other system resources, as described later. For instance, a disk device may offer more than one operation with different service times, so the disk is modelled as a task with operations modelled by entries, plus an underlying device for the actual execution.

Figure 2 shows one graphical LQN notation. The tasks (software servers) are the large parallelograms, and their entries are the enclosed parallelograms. The task of Figure 1 could for instance be the `HTTPServ` task, with its thread pool indicated by the multiplicity notation $m = 20$ for a limit of 20 threads. One `webService` request requires a total of 15 ms (CPU time) to handle it, including receiving the request, interpreting it, and making a nested request to the `Application`, receiving the response, and sending the final response page back to the client. The pool of 150 clients is represented by the single `Client` task symbol, with an entry that represents one cycle of client activity: think for 2 seconds, make one request. It runs on a single processor. The request arrows to the `Application` entries show that 90% of requests are to the `Browse` entry, and 10% to the `Buy` entry. The host demand of the entries is 30 ms for `Browse`, 250 ms for `Buy`, and `Application` has 10 threads and runs on two cores (or a two-CPU multiprocessor). The host demand labels indicate only one phase of service at each layer in this model.

A real system may have additional layers. If this is an e-commerce system, the `Application` will make requests to one or more database systems (which will in turn use a storage system), and to some kind of financial server for the purchase transaction. An HTTP server usually has separate storage for fixed page elements and makes requests to that.

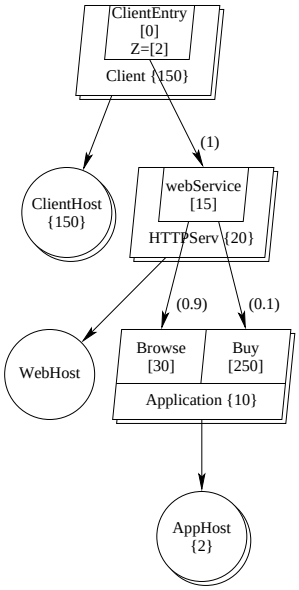


Figure 2: A layered application with software and hardware servers.

Tasks, entries, calls, demands

The notation for layered queueing models uses the terms task, host processor, entry, call, and demand, as follows.

- *Tasks* are the interacting entities in the model. Tasks carry out operations defined by their *entries* and also have the properties of resources, including a queue, a discipline, and a multiplicity. A single-threaded task is itself a critical section.

  Tasks that do not receive any requests are special; they are called *reference tasks* and represent load generators or users of the system, that cycle endlessly and create requests to other tasks. Separate classes of users are modelled by separate reference tasks.

- A task has a *host processor*, which models the physical entity that carries out the operations. This separates the logic of an operation from its physical execution. The processor has a queue and a discipline for executing its tasks, and a task has a priority. The host processor has a speed ratio relative to a "standard processor", such that when the host demand s of an entry given for the standard processor, and the actual demand is s/(speed ratio).

  Thus a disk is modeled by two entities, a disk task representing the logic of disk operations (including the difference between say a read and a write, and the logic of disk device-level caching), and a disk device.

- *Calls* are requests for service from one entry to an entry of another task. A call may be synchronous, asynchronous, or forwarding, as discussed in Section 5. This gives a rich vocabulary of parallel and concurrent operations, expressed in a manner very close to that of a software architecture description language.

- *Demands* are the total average amounts of host processing and average number of calls for service operations required to complete an entry. More detailed descriptions, detailing the sequence of op-

erations, can be given by giving the *activity structure* of an entry or a task, which will be described below.

The basic use of these elements to model software tasks is illustrated in Figure 2. Extensions of the modeling concepts will be described later, including: defining parallel operations with activities, using a task to model pure delays (such as a network latency), using a task to model other kinds of software resources such as semaphores, exclusive locks, or buffer pools, and using forwarding interactions to model asynchronous operations. A large family of replicated systems can be modeled compactly using replication.

2.1 Model workload parameters

The software workload is described by the parameters of an activity. The basic model of an entry assumes it has a single activity (sometimes called "phase one" above); some systems have a second activity (phase two) after sending a reply. More generally, a graph of activities may be defined to describe the execution of an operation (see section 6). The parameters of an activity are:

- execution demand: the time demand on the CPU or other device (indicated by the parameter labelled "`s`" in the modeling code)
- wait delay (also called a *think time*) (optional... it can be used to model any pure delay that does not occupy the processor) (parameter label is "`Z`")
- mean synchronous requests to another entry (token label is "`y`")
- mean asynchronous requests to another entry (parameter label is "`z`")

Additional optional parameters of an activity are:

- the probability of forwarding the request to another entry, rather than replying to it, when serving a synchronous request (parameter label is "`F`", there can be multiple probabilities)
- the squared coefficient of variation of the execution demand requests. This is the ratio of the variance to the square of the mean; for a deterministic demand its value is 0 (parameter label is "`c`")
- a binary parameter to identify a *stochastic sequence* in which the number of requests from the activity is random, with a geometric distribution and the stated mean, versus a *deterministic sequence* in which the number is exactly the stated number (which must be an integer). (the parameter label is "`f`", with value 0 for stochastic (the default) or 1 for a deterministic sequence)

2.2 Performance measures: service time and utilization of an entry or a task

The *service time* of an entry, shown in Figure 3, is the time it is busy, in response to a single request. It includes its execution time and all the time it spends blocked, waiting for its processor and for nested lower services to complete. The service time in a layered model is a result rather than a parameter, except for pure servers such as hosts.

Since a task may have entries with different service times, the entries define different classes of service by the task. The overall mean service time of a task is the average of the entry service times (weighted by frequency of invocation).

The *utilization* of a single-threaded task is the fraction of time the task is busy (executing or blocked), meaning not idle. A multi-threaded or infinite-threaded task may have several services under way at once and its utilization is the mean number of busy threads.

A saturated task has all its threads busy almost all the time.

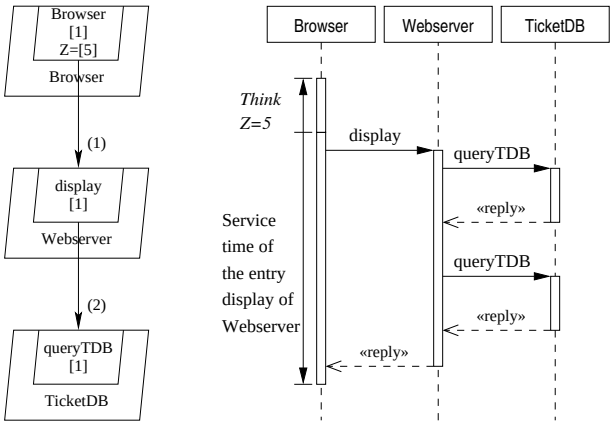


Figure 3: Service time of an entry, indicated on a sequence diagram.

Software Bottlenecks

When a task is fully utilized but the resources it uses (by nested calls, or its processor) are not fully utilized, the task is called a "software bottleneck". A typical example is a single-threaded task which blocks for I/O, and the typical cure is to multi-thread the task. Then when one thread is blocked another one can proceed. (See the paper [**?**] on software bottlenecks).

3 Tools: the LQNS solver, the simulator, and their modeling language

There are two solvers for layered queues, that take the same input files:

- The analytic solver *lqns* can be executed with the command line:

  ```
  lqns infile.lqn
  ```

  (where the suffix `.lqn` is suggested, but not mandatory)[1] and it produces an output file with the default name `infile.out`. There are many options, which can be listed by invoking `lqns -h`. The documentation is the manual ("man") page, also available in ASCII as `lqns.txt` or PDF as `lqns.pdf`

  The simulator *lqsim*] is invoked as:

  ```
  lqsim [run controls] infile.lqn
  ```

  and also generates an output file `infile.out`, which includes confidence intervals on the estimated performance parameters. Documentation is the man page for lqsim, the ASCII version lqsim.txt, or PDF as lqsim.pdf

There are also two languages for defining models, a cryptic "original" language and an XML-based language LQML which is now the normative language. A conversion program `lqn2xml` is used to convert

[1] Input files with suffixes of `.lqn`, `.in` or `.txt` are processed using the original input grammar. All other suffixes are treated as XML input.

between them. Since LQML is very verbose, models are often written in the original language and converted before running, or run from that. Another approach is to use the semigraphical editor jLQNDef. This editor can be used from scratch or to display and edit models created earlier, and it can read and write in either syntax.

LQNS includes an experiment control section written in a language called LQX (for LQ eXperiments) which can set parameters to various values, extract and format results, and force new solutions based on the results of a previous solution. LQX can only be used with LQML input.

Simple experiments involving solutions over sets of parameter values can also be run with LQNS/LQSIM in a simpler format using the "original" language plus annotations for parameters and results. This facility replaces the experiment controller SPEX, which worked with the original language but has become unmaintainable. See section 11.1.

4 Task resources, queueing, and multiplicity (threads)

A task models those properties of a software object which govern contention for executing the object; it is identified with a *task resource*, and a task queue. One task queue is used by all requests to all the task's entries. Typically the software object is an operating system process, but tasks can also represent other objects.

A *single-threaded* task can respond to one request at a time. A *multi-threaded* task has a limited pool of instances, each of which can respond to a request (it acts as a queueing multiserver). The instances are homogeneous and are dispatched from a single queue, like any multiserver. Multiple instances may be provided in different ways, for instance by process forking, by a static pool of lightweight threads or kernel threads, or by re-entrant procedure calls. A special case which can be modelled as multiple instances is a single carefully written process that accepts all requests, and saves the context of uncompleted requests in data tables (this is called *virtual threading*, or *data threading*)

Some software objects are fully re-entrant and can exist in any number of copies, each with its own context. These are often called multi-threaded, however because there is no limit to the number of threads we will term them *infinite-threaded*. They impose no resource constraint, and are used to represent pure delays.

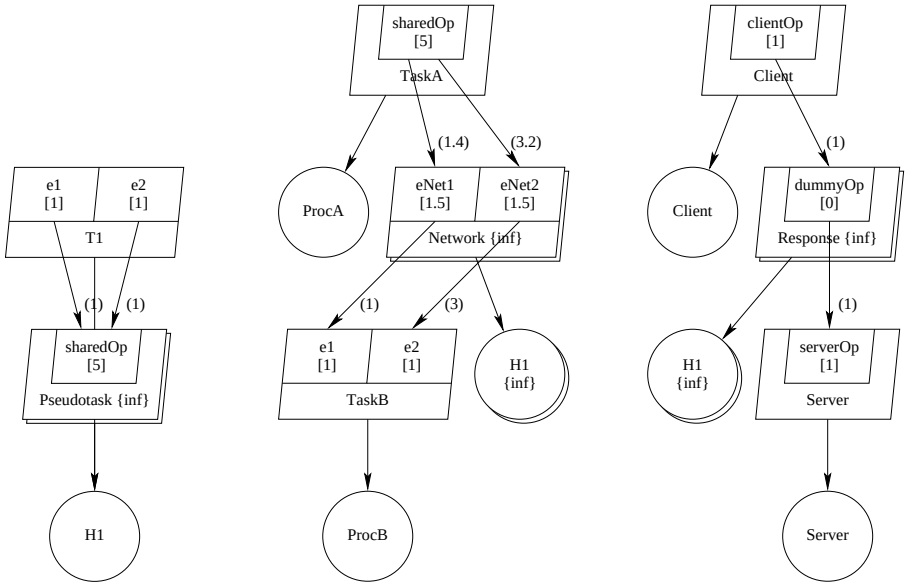
Multiplicity also applies to processors (hardware servers) which may be defined as single, multiple, or infinite. An infinite processor is an artificial entity introduced to provide a host for an infinite task (every task must have a host). If there are several infinite tasks, they can all be hosted on a single infinite processor.

4.1 Driver tasks and arrival processes

Arrivals to the system are modeled by tasks which generate requests spontaneously, called *reference tasks* in LQN. Any task with no requests made to it is a reference task. It should have a single entry with just one phase. Each reference task generates a separate class of requests.

A closed class with population N and think time Z is represented by a reference task with multiplicity N and think time Z, hosted by a processor with multiplicity at least N, and making synchronous requests (that block waiting for a reply). The importance of the blocking is that the client task cannot do anything else while the response is coming back, thus imposing the finite population of simultaneous requests.

It is usually preferable to model open arrivals by a closed arrival approximation (see below) However, an open class with arrivals at rate f/sec can be modeled by a reference task of multiplicity 1 and think time $Z = 1/f$ sec, and making an asynchronous call to some operation. Since the call is asynchronous, the task immediately begins to generate the next call, giving the arrival process. If the think time is exponentially



(a) a shared module (b) a network latency (c) instrumentation

Figure 4: Pseudotasks

distributed (the default), and the entry has a single deterministic phase, the arrivals will be Poisson. As an alternative way to model open arrivals, any entry can be defined to have a Poisson stream of arrivals coming to it, in addition to other requests.

Closed modeling of open arrivals

Since the service time of entries in layered queueing is not known in advance (it may include lower-layer contention delays), it is not possible to guarantee that a given arrival rate does not over-saturate the system. The analytic solver and simulator detects this condition during iteration and stops. For this reason it is better to make a closed approximation to the open stream. Introduce a reference task with a very large population N (large relative to the expected queue lengths), and a large think time $Z = N/f$, to get an approximate rate f. If N is not large enough, the rate will fall, and this is a signal to increase N (or, that the system cannot handle the rate f/sec).

4.2 Pseudo-tasks

Artificial pseudo-tasks can be used to model a variety of things, including an internal operation of a "task", a network delay, or a response-time measurement probe. They may be hosted on pseudo-processors.

An internal operation of a task

Refer to Figure 4a. A LQN task `T2` can model a re-entrant module that has no attached resource significance, for example a module within a task `T1` that is shared by several of its entries. Instead of including the workload of the module in all the entries, it can be modelled separately (to better reflect the software structure, or just to calculate its loading within the model). The pseudo-task `T2` is an infinite server that runs on the same host as the task `T1`. Since it is always called by an entry of `T1`, its task resources are provided by `T1`.

A network delay

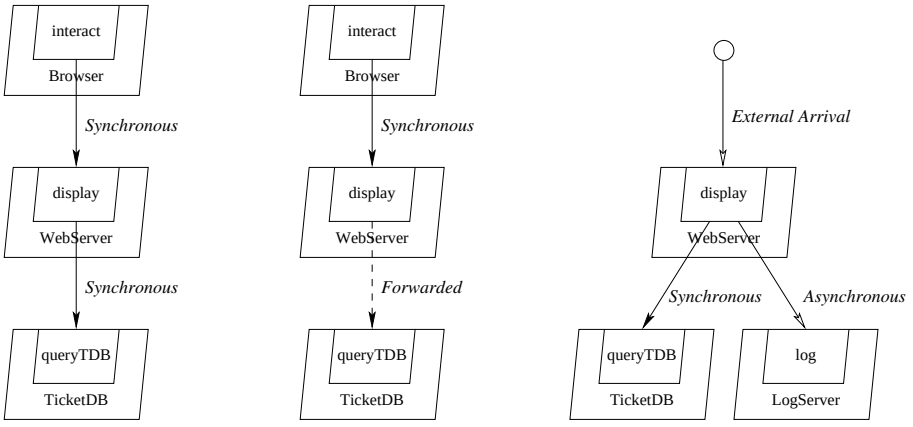
Refer to Figure 4b. An infinite task with host demand equal to the delay time can be interposed in any call to impose this delay on the call. Multiple calls over the same network to different destination entries must have separate entries in the delay pseudotask, to separate the call routing. The task is hosted on an infinite pseudo-processor of speed factor 1.0.

Response-time instrumentation

Refer to 4c. The LQNS solver returns the delay to a call (such as the call from `clientOp` to `serverOp` in Figure 4c) in two parts, as the queueing time of the request at `Server`, plus the service time of the entry `serverOp`. Purely for convenience, we sometimes define a dummy pseudotask that simply passes on the call; the task "Response" blocks for a time equal to the queueing plus service at serverOp. "Response" does not have to be infinite, but it and its pseudo-processor host must have equal or greater multiplicity, than `Client`.

5 Styles of interaction

We can model three kinds of interactions between entries which are illustrated by a sequence diagram in Figure 6 matching the LQN diagram in Figure 5 for each interaction.



(a) Nested synchronous calls

(b) Forwarded query, replying directly to the client

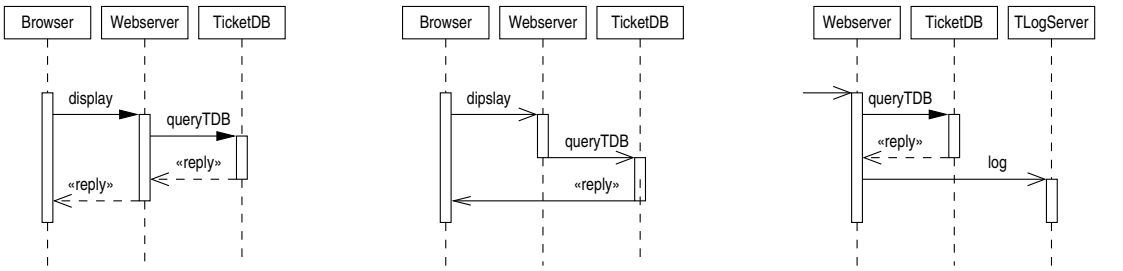
(c) External arrivals, and an asynchronous invocation

Figure 5: Layered Queueing notation for the three styles of interaction.

- *asynchronous call:* the sender does not wait and receives no reply. The receiving entry operates autonomously and handles the request.

  This propagates the workload among the servers and gives the resource utilizations and throughputs, but it is an effort to determine the delays along the path taken by a request. For this reason an asynchronous chain of operations may be modeled by forwarding.

- *synchronous call*, with a reply: The sender waits (blocked) for the reply, and the receiving entry must provide replies. This is the pattern of a standard remote procedure call (RPC). The sending object task resource or thread is regarded as busy during the wait.



(a) Nested synchronous calls

(b) Forwarded query, replying directly to the client

(c) External arrivals, and an asynchronous invocation

Figure 6: UML Sequence Diagrams showing the messages passed. In this notation, the solid arrowhead shows a synchronous message, with a dashed arrow for the reply.

For non-blocking software the request-reply structure may be kept in the model by introducing artificial sender threads, one for each potential outstanding reply. In the model these threads do block and wait for the reply, and they accumulate the total time to complete a response. These threads need not exist explicitly in the software.

Some "delayed-synchronous" interactions with a reply actually continue in parallel with the execution of the server, and later attempt to pick up the reply from an agent or mailbox. This can be modeled as parallel execution, with activities (see section 6).

- *forwarding interaction:* the first sender sees a synchronous interaction, and waits for a reply. However the receiver does not reply, but rather forwards the request to a third party, which either replies, or forwards it further. This gives an asynchronous chain of operations for a waiting client. Further, at each step there may multiple forwarding targets, with probabilities.

  Asynchronous sequences of events can be modeled by forwarding, in order to structure the sequences in the model and to capture the delay over a path. If the sender does not block, an instrumentation pseudotask (infinite-threaded) can be introduced to at the beginning of the path, which forwards the request, receives the final "reply" when the sequence ends, and records the delay as the service time of the waiting thread.

6 Adding detail with activities within an entry

The simplest operation by an entry is a single activity, followed by a reply to the invocation. However it is important to be able to specify more complex behaviour patterns, including operations in a particular sequence, and parallel subthreads. This is done by specifying a precedence graph of activities, starting from the first activity for the entry.

Each activity is described by a set of parameters with labels "`s`", "`y`", "`f`", "`c`" etc., as described in Section 2. The graph can then be described by a series of textually specified relationships:

Sequence:	`a1 -> a2`	activity 1 preceedes activity 2
AND-fork:	`a1 -> a2 & a3`	activity 1 precedes activities 2, 3... in parallel (an AND-list of any length)
AND-join:	`a1 & a2 -> a3`	predecessors are an AND-list of any length.
OR-fork (branch):	`a1 -> (p2) a2 + (p3) a3`	
OR-join (merge):	`a1 + a2 -> a3`	predecessors may be an OR-list of any length.
LOOP:	`a1 -> n2 * a2,  n3 * a3, a4`	a repeated activity 2, and a repeated activity 3, followed by activity 4.

If the repeated part is not a single activity then:

- if it is a sequence, the rest of the sequence is defined separately, with a definition that it is preceded by activity1.
- If it is a complex structure of activities it can be packaged into a separate pseudo-task (as described in Section 2.1), called by activity1. This pseudo-task can contain forks and joins and other behaviour structure nested within the loop.

Figure 7 shows an example, including both ways to structure a loop. The reply is sent after `a10`. The loop defined by the call from `a14` is defined in the workload parameters of `a13`, not in the precedence graph. The parallel activities `a7` and `a8` are executed by two concurrent threads which compete for the host processor; one or both are assumed to be created dynamically at the fork and destroyed at the join. Physical parallelism is obtained when the host is a multiprocessor, or if one or both of these activities call other tasks on different hosts.

Although there is commonly a separate sub-graph for each entry, the activity graph is a property of the task as a whole. The reason is, that it allows one to define a single graph with multiple starting points at two or several entries, for instance to define a task which joins flows from two different tasks.

Parallelism in layered queues is discussed in [**?**].

6.1 LQN code for the activity section

LQN code for the activity section The textual definition above is part of the "original" LQN language; the XML-based LQML has a more extended syntax.

In the LQN language, the entries which use activities are identified in the entry list, along with the first activity in the entry. In a separate activity section for the task, the workload parameters of the activities and their precedence relationships are defined for all the entries that have activities. Also the workload of each activity is defined, using the parameters defined in Section 3.1. The template file `activity-templ.lqn`, repesenting a server with OR and AND forks, is commented to document the syntax including that for activities.

6.2 Modeling Asynchronous RPC, and Prefetching

An asynchronous RPC is modeled by forking an activity to make the RPC, and joining at the point where the result is picked up by the main flow. Prefetches are modeled similarly, as are "futures" operations (which do a speculative computation in parallel).

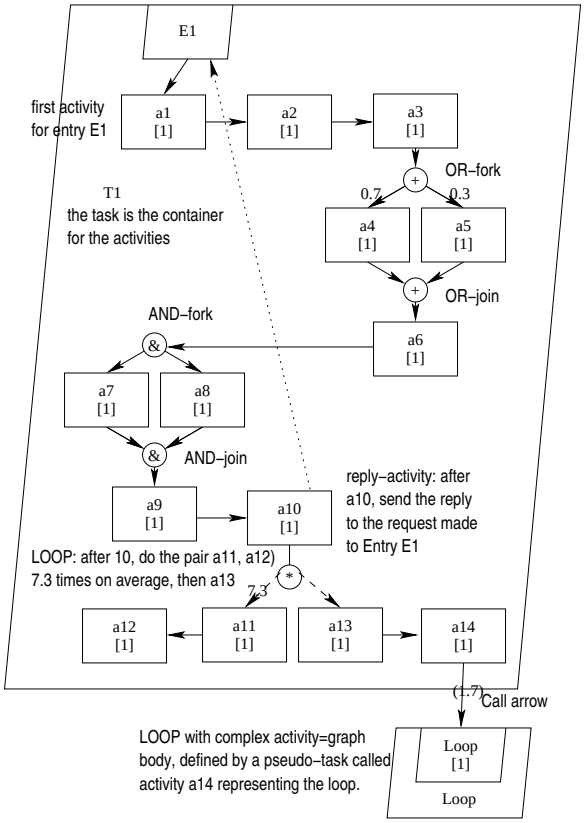
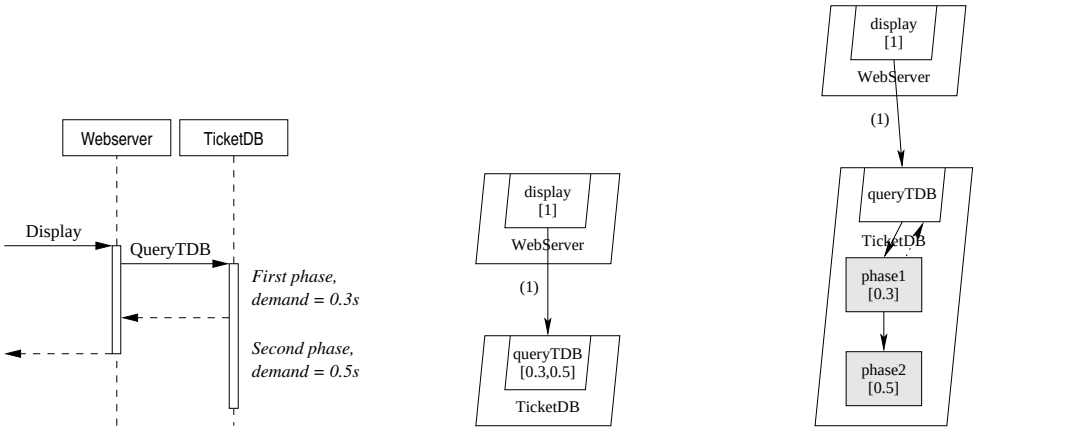


Figure 7: Task with activities

7 Service with a Second Phase

A wide variety of software services give a reply to the requester before they are entirely finished, in order to release the requester from the synchronous call delay as soon as possible. The remaining operations after the reply are done under sole control of the server task, and they form the second phase. This behaviour is shown in Figure 8. A special shorthand is used to represent this common case.



(a) Behaviour (b) LQN with phases in TktD (c) LQN with activities in TktDB (activities shaded)

Figure 8: A second phase of service lets the client of the interaction proceed

Entries with phase one and phase twoa can be represented by two activities, one performed before the reply and one after. Because they all have this simple structure they can be defined directly for the entry, without an explicit precedence graph. Each entry has a vector for each workload parameter, with a value for each phase. Thus the execution demand for the entry `queryTDB` above would be defined by a line beginning with code "s" for execution demand:

```
s queryTDB 0.3 0.5 -1
```

The host demands are 0.3 for phase one, 0.5 for phase two. Note that the separator `-1` is used in many places in the definition language.

Second phases are common. An example is seen in a write operation to the Linux NFS file server; the write operation returns to the requester once the file is buffered in the server, and the actual write to disk is performed later, under sole control of the server. Doing the writes in first phase would be safer, because the client would be told if the write failed, and this is how the NFS protocol was originally defined. Other NFS implementations allow second-phase or delayed writes only if the server has a battery-powered buffer to provide security of the data, in case of a power failure.

Second phases improve performance; they give less delay to the client, and they increase capacity because there can be some parallel operation between the client and the server. The amount of gain depends on circumstances (real parallelism needs separate processors, and a saturated server cannot increase its capacity).

The extreme case of all execution being in second phase is a kind of acknowledged hand-over of data from the client to the server. Thus it is similar to an asynchronous message, except that the sender waits for the acknowledgment. One important advantage of this is that the sender cannot over-run the receiver; the sender is throttled by the waiting for acknowledgments.

Results for second phase at a single server, and at two layered servers

If some of a task's work can be put into the second phase, the task can return more quickly to its clients. However if the task is already saturated the clients have to wait for many other services anyway and the advantage is small or even nil. The degree of improvement thus depends on the degree of saturation, and where the saturation is. Table 1 shows how a group of users with a 5-sec thinking time are affected when the server service time is split between phase one and phase two in different ratios (LQNS approximate results). At low utilizations, more phase two is uniformly better, but as utilization increases, the best split moves towards the middle.

nusers	Response time (sec) for different values of demand s = [phase 1, phase 2]					
	s = [0,1.0]	[0.2,0.8]	[0.4,0.6]	[0.6,0.4]	[0.8,0.2]	[1.0,0]
1	0.166	0.310	0.464	0.629	0.807	0.999
4	1.125	0.726	0.827	0.996	1.269	1.6420
7	3.066	2.694	1.5087	1.7928	2.269	2.9256
10	4.8474	4.2741	3.8951	3.98149	4.4927	5.221
15	9.9096	9.4738	9.2281	9.2289	9.5048	10.037
20	14.9381	14.541	14.323	14.318	14.548	15.0120

Table 1: The impact of dividing a unit server demand between phase 1 and phase 2

8 Logical resources (critical sections, locks, buffers)

The task entity in layered modeling is used to model any resource whatever. Consider first a critical section shared by a set of concurrent threads or processes. There are two cases:

- if the threads or processes are on the same processor, and they all execute the identical operation in the critical section (this is like a monitor) then the operation can be removed from the tasks and made into an entry in a pseudo-task with multiplicity 1. This is like a pseudo task for an internal operation, but imposing concurrency restrictions.

- if the threads or processes are on different hosts, or they execute different operations in the critical section, then the pseudo-task with multiplicity 1 runs on a psedo-processor, with an entry for each caller... this entry calls a sort of shadow task defined for each process, which represents the critical section workload of that process. The shadow tasks are defined as a "task for an internal operation" described above, with multiplicity 1, and run on the same host as the caller. Only one of these shadow tasks can be active at once, and a queue of requests forms before the critical section "task". Effectively the calling tasks are split into a task for the workload outside the critical section, and a task for the inside.

The same approach can be applied to locking a table, with the operations applied while holding the lock, treated as a critical section. A complex locking system, with many separate locks, and read and write locks, requires special treatment. The queuing disciplines are complex, and there are often too many locks to represent each one separately. This is a subject of current research.

Memory and buffer resources can be modeled similarly, with a multiple "task" (the same designation as a multi-threaded task) to represent the resource, and with entries to activate the workload for each user.

Modeling finite buffers:

A finite buffer space, which blocks its senders when it is full, can be modeled by a multi-threaded buffer task with second-phase interactions going into and coming out of the buffer. The model is shown in Figure 9. Each buffer space is modeled by a *"thread"* which immediately replies (releasing the sender) and then sends to the receiver (and waits until the receiver replies, to acknowledge receipt).

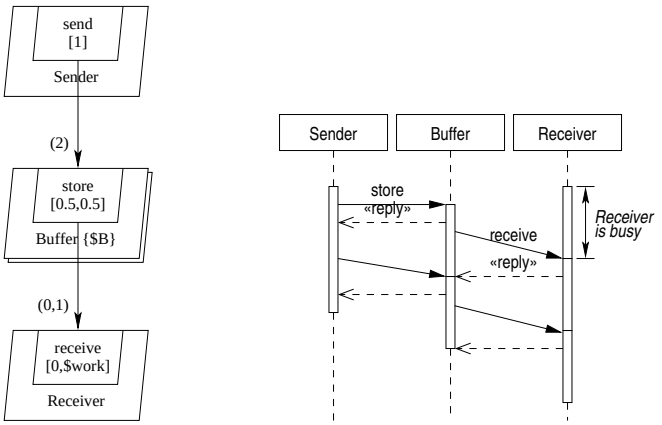


Figure 9: Buffer pseudo-task.

9 Limitations

This is not a complete list, but notes some limitations that have come up:

- recursive calls are excluded (a task calling its own entries)... the approach for dealing with recursive calls is to aggregate the demand into the first entry. Possibly this should be accommodated, but it requires an assumption that the same thread handles the recursive request (to avoid deadlock when threads are limited), which limits the behaviour of an entry.
- replication of subsystems (without defining all the replicas separately) is handled but only with restrictions; a thesis is available [**?**].
- activity sequences which fork in one task and join in another can be solved by the simulation solver (lqsim).
- external arrival flows may be specified into an entry, however a system with only external arrivals causes problems for the analytic solver. It is recommended to define sources of traffic as tasks which make requests and block; this has the advantage that it never overruns the system. Message loss is not modeled; current research may cure this.
- exceptions and timeouts are also not modeled, similar comment.

10 Reference material

Also, see the web pages:

- http://www.layeredqueues.org/ includes a bibliography on layered queueing.
- http://www.sce.carleton.ca/rads for material on the larger project (RADS is the Real-time And Distributed Systems group at Carleton), and for software download.
- http://www.sce.carleton.ca/rads/lqn/lqn-documentation
- http://www.sce.carleton.ca/faculty/woodside for my bibliography material.

11 Running the tools

LQNS is available from the download page (you will need to sign a license and fax it) for Linux, Windows and MacOSX. It has a comprehensive man page describing many options for different solver algorithms, for tracing solutions and for printing more or less detail. The best reference on the many solver options is Greg Franks' PhD thesis [**?**].

The LQNS input language is essentially documented by the comments in the example files found ni the Appendicies. There is a BNF definition included in the more extensive discussion of activity notation, in "Parallel Activity Notation in LQNS" by Franks (Chapter 6 in the thesis [**?**]). Personally I create models starting with one of two template files that include example definitions and comments defining the syntax. `template.lqn` does not include activities, `activity-template.lqn` does. I replace the model in the template by the one I want.

LQML is the more recent XML-based language, with schema `lqn.xsd` (and there are two sub-schemas with it). You can convert between the two languages with the program `lqn2xml`, or the `jlqndef` editor. If you need the XML it may be preferable to create the model in the input language and convert, writing models in XML is brutal.

`lqn2ps` is a program for creating a graphical image in many formats, not just postscript.

`Jlqndef` is a graphical and text-window-based editor which shows a simple diagram of the model. It requires Java 1.1.3 or higher, with the "swing components".

11.1 Efficient experiment control

Two ways are provided to do parameter sensitivity and other experiments under program control, and extract the results in a compact form.

1. The most general is to write a progam in LQX and include it in a LQML definition file (see the LQNS User Guide [**?**])
2. A more restricted but useful capability allows one to define sets of values of parameters, and then evaluate models for all combinations of these values. This replaces SPEX. It is provided by `lqns` or `lqsim` using the original LQ language with SPEX-like additions to the model file, which are described in the User Guide. If you are familiar with SPEX, the main difference is that the parameter range definitions must be enclosed in square brackets (e.g. `$param = [1,6,15,37]` or `$param = [10:100,10]`. Also expressions must conform to LQX rather than PERL, comments must begin with #, and string substitution is not supported, and there is a change to how solver pragmas are set.

   If you are not familiar with SPEX, a simplified description of experiment control follows (for more capabilities and details see the User Guide, section 5.3):

- before the model begins, add lines of form `$paramname = [value1, value2,...]` or `$paramname = [start:end,increment]` (where *paramname* must begin with $) to define the experiment cases.
- optionally you can define other named parameters as functions (expressions) of these, using +, − ⋆, / and ⋆⋆ operators in the usual way.
- within the model, use `$paramname` in place of the numeric model parameter value.
- add results definitions at the ends of lines defining processors, tasks, entry host demands and calls. Each result definition is of the form `%X $resultname`, where *resultname* also begins with $ and `X` is a code, one of:

  u: (utilization),

  pu: (processor utilization),

  s0: (total entry service time),

  s1: (entry phase one service time),

  s2 (entry phase 2 service time), and

  f: (entry or task throughput).
- after the model, add a section starting with "`R 0`" and ending with "`-1` with a list of items:

  (a) parameter names,

  (b) result names,

  (c) `$var = expression`,

  one per line with no end of line termination. Each case solved will give one line of results with these values in the order defined. The special parameter `$0` is the model number created by the solver, starting from one.
- The results from the solution will be directed to the terminal by default. They can be redirected using "`-o` *filename*".

12 Questions

- *Throughput refuses to increase when I introduce more resources.* Search for a saturated resource; it may represent a modeling error. For instance, if one introduces more users, one must also introduce more processors for them to run on. A simple expedient is to make any resource which should not be a limit, infinite. This is also solver-friendly, a infinite resources are easy.

- Convergence: *what if my LQNS solution does not converge?* The symptom is that the convergence value for the iterations is greater than the set value, typically 10^{-6}. Sometimes, especially in heavily loaded systems, the iterative solver will cycle and not converge. One cure which is sometimes effective is to reduce the value of the under-relaxation coefficient in the solver controls, from a typical value of 0.9 or 0.7 to a low value, of say 0.1. This is intended to force convergence by reducing the step size. If this does not succeed, then as long as the convergence value is less than 0.1, the solution found has some reasonable relationship to the correct value. (The size of the convergence value is the largest relative change from one iteration to the next, of any of the variables in the model; it does not directly indicate the size of errors, but if the solution is in fact cycling around the correct solution then all relative errors are probably smaller than this). A method which is usually not effective is to increase the number of iteration steps. The basic recourse for greater accuracy is to simulate.

- Replication: *how can I model a system with many repetitions of a subsystem within it?* Provided the replications are identical in every respect, and divide all their interactions with the rest of the system equally, the replication feature of LQNS can give efficient solutions. The full documentation of this feature is in the Master's thesis of Amy Pan. Briefly, any task can have a replication factor r, which means that multiple copies are created. If its processor has the same r, then each copy has a separate processor. If it communicates with other tasks with the same r, it communicates with just one of them, and the interactions are assumed to form r subsystems. Messages between tasks with different r must have values of fan-in "`i`" and fanout "`` `o' `` such that the product of source ($r \times$ fanout) = destination ($r \times$ fan-in). These factors `i` and `o` describe the replication of the message arcs.

- Odd results for multiple servers: *if I run for a series of values of m for server multiplicity, I may see rising throughput; then for $m = \infty$, the throughput drops a bit. How come?* The waiting time calculation for a multiserver is an approximation, and errors of a few percent are to be expected. The infinite server queue is solved exactly (no waiting). If the anomaly is worrying, try a more exact multiserver algorithm by using "pragma -Pmultiserver=conway", but it will take longer.

- Non-blocking systems: *in my model the servers are asynchronous. A server processes each message, whether a request or a reply, and then takes whatever message comes next. I never blocks to wait for a reply. How to model it?* Such a server is modeled with infinite threads, allowing one active thread for each uncompleted request it is processing. This may be called virtual threads or data threads , since the request context, if any, is managed by user data.

- Solution Time: *my LQNS solution takes a long time (a minute is long for a few tasks; 10 minutes is long for any model)*. Possible reasons are:

  1. poor convergence (see below)... you may want to reduce the iterations or simulate;
  2. a huge number of classes, generated by having a lot of separate source tasks ("reference tasks" and lots of entries on worker tasks.... you could get faster results if the sources were combined into fewer tasks, with random splits to generate the requests they make into the service layers.
  3. do you have a multiserver (not a reference task) with a large m (say, $m > 20$?)... or layered multiservers one above the other, with moderate m... multiserver solutions are only moderately expensive by the default algorithm, but the others cost more. You might consider whether it could as easily be infinite (if, say, its usage is well below the limit so the limit is not a factor).

  In any of these cases, you might try to simulate; there have been models that solved faster by simulation.

- Cycles: *what do I do if my model has a synchronous messaging cycle?* LQNS will refuse to solve a model, however it can be instructed to ignore the cycle checker with the `pragma cycles=allow`. It then solves the model with an implicit assumption that if deadlocks are possible, they do not occur or are resolved. The simulator (lqsim) will take a model with cycles at any time, however if a deadlock occurs as a result of a cycle, the simulation stops without any diagnostic.

- Delay: *how can I get the result for delay from an input at one point, to a response completion somewhere else in the model?* The cleanest approach is to introduce some special model elements: first, at the start of the response, introduce a pseudo-task R to capture the delay as its service time. It has

zero demands (s = 0), has infinite multiplicity (code i) runs on its own processor or an infinite processor, has deterministic phases (f = 1) and makes one synchronous call to the input point to start the response. Second, create a forwarding chain through the model along the path of the response, so that the reply is created at the completion point; the reply goes back to the pseudo-task R, and ends its blocking state.

- Simulation accuracy: *how can I tell how accurate my simulation is?* It is essential to get confidence intervals out of the simulation. If you don't know about these, you will have to consult a statistics text. Lqsim will calculate confidence intervals for you, for all its results, if you run with the `-A` (automatic) or `-B` (batched) run options. These have the form `-A,a,p,b` or `-B,n,b` where b is a batch length in model time units, which should be say 100 times longer than the longest service time in your model, and n is the number batches (suggest 30, which is the max allowed). For A, a is the accuracy target in 5 of mean values, p is the confidence level.

13 Model File: reserve-templ.lqn

```
# SRVN Model Description File, for file: reserve-temp.lqn

G
# Comments between quotes, as many lines as necessary
"Layered Queueing Network for a Web-based Reservation System"
0.0001 # Convergence criterion,
500 # iteration limit,
1 # print interval,
0.5 # underrelaxation (stabilizes the algorithm if less than 1.)
# End of General Information
-1

# Processor Information (the zero is necessary; it may also give the number of proc
P 0
# SYNTAX: p ProcessorName SchedDiscipline [multiplicity, default = 1]
# SchedDiscipline = f fifo
# | r random
#  | p premptive
#  | h hol (or non-pre-empt)
#  | s proc-sharing
# multiplicity = m value (multiprocessor)
# | i (infinite)
 p UserP f i
 p DBP f
 p ReservP f
 p CCReqP f i
 p ReservDiskP f
 p DBDiskP f
-1

# Task Information: (the zero is necessary; it may also give the number of tasks)
T 7
# SYNTAX: t TaskName RefFlag EntryList -1 ProcessorName [multiplicity]
# TaskName is any string, globally unique among tasks
# RefFlag = r (reference or user task)
# | n (other)
# multiplicity = m value (multithreaded)
# | i (infinite)
 t Users r users -1 UserP m 100
 t Reserv n interact disconnect connect -1 ReservP m 5
 t Netware n netware -1 ReservP
 t DB n dbupdate -1 DBP
 t CCReq n ccreq -1 CCReqP i
 t ReservDisk n reservDisk -1 ReservDiskP
```

```
 t DBDisk n dbDisk -1 DBDiskP
-1

#Entry Information: (the zero is necessary; it may also give the total number of er
E 0
# SYNTAX-FORM-A: Token EntryName Value1 [Value2] [Value3] -1
# EntryName is a string, globally unique over all entries
# Values are for phase 1, 2 and 3 (phase 1 is before the reply)
# Token indicate the significance of the Value:
# s HostServiceDemand for EntryName
# c HostServiceCoefficientofVariation
# f PhaseTypeFlag
# SYNTAX-FORM-B: Token FromEntry ToEntry Value1 [Value2] [Value3] 1
# Token indicate the Value Definitions:
# y SynchronousCalls (no. of rendezvous)
# F ProbForwarding (forward to ToEntry rather than replying)
# z AsynchronousCalls (no. of send-no-reply messages)
# o Fanout (for replicated servers)(ignore this)
# i FanIn (for replicated servers)(ignore this)
# This example only shows use of host demands and synchronous requests
# ---------- Users ----------
 s users 0 56 -1
 y users connect 0 1 -1
 y users interact 0 6 -1
 y users disconnect 0 1 -1
# ---------- Reserv ----------
 s connect 0.001 -1
 y connect netware 1 -1
 s disconnect 0.0001 0.0007 -1
 y disconnect netware 1 0 -1
 y disconnect dbupdate 1 0 -1
 s interact 0.0014 -1
 y interact netware 1 -1
 y interact ccreq 0.1 -1
 y interact dbupdate 1.15 -1
# ---------- Netware ----------
 s netware 0.0012 -1
 y netware reservDisk 1.5 -1
# ---------- DB ----------
 s dbupdate 0.0085 -1
 y dbupdate dbDisk 2 -1
# ---------- CCReq ----------
 s ccreq 3 -1
# ---------- ReservDisk ----------
 s reservDisk 0.011 -1
# ---------- DBDisk ----------
```

```
  s dbDisk 0.011 -1
-1
```

14 Model File: activity-templ.lqn

```
# SRVN Model Description File, for file: activity-templ.lqn
# This template documents the use of activities in depth
# It assumes familiarity with other features documented in
# reservtemplate.lqn or in template.lqn

G "Activity template" 1e-06 50 5 0.9 -1

P 4
 p UserP f i
 p ServerP s 0  #processor sharing at the server
 p Disk1P f
 p Disk2P f
-1

T 5
 t User r user -1 UserP z 50 m 50
 t Server n server -1 ServerP m 4 # 4 threads (with activities)
 t BigLoop n bigLoop -1 ServerP i # pseudo-task for a complex loop pattern
 t Disk1 n disk1read disk1write -1 Disk1P
 t Disk2 n disk2read disk2write -1 Disk2P
-1

E 7
# ---------- User ----------
 s user 1 -1
 f user 1 -1
 y user server 1 -1  # one request to the server per cycle.
# ---------- Server ----------
 A server serverStart  # Entry server is defined using activities.
# ---------- BigLoop ----------
 A bigLoop first
# ---------- Disk1 ----------
 s disk1write 0.04 -1  # operation time of this entry.
 s disk1read 0.04 -1
# ---------- Disk2 ----------
 s disk2write 0.03 -1
 s disk2read 0.03 -1
-1

#Optional sections for definition of activities
# One section for each task that has activities, beginning A TaskName
# list of activity parameters, using the syntax for entry parameters, but
# with just one value and no terminator -1
# : (separator), then a section for precedence among activities
```

```
# Syntax for precedence:
# a1 -> a2 for sequence
# a1 -> a2 & a3 ... AND-fork (any number)
# a1 & a2 ... -> a3 AND join
# a1 & a2 ... -> a3 & a4 ... AND join followed by AND fork
# a1 -> (prob2)a2 + (prob3)a3 ... OR fork (any number, with probabilities)
# a1 -> meanCount*a2,a3....for a repeated activity a2, followed by a3
# (notice that activities that follow a2 are inside the loop)
# a6[entryName] indicates that after a6, a reply will be sent to entryName
#
A Server
  s serverStart 0 # Every activity that is used must have a host demand
  s seqInit 0.3
  s parInit 0.1
  s parA 0.05
  y parA disk1read 1.3
  s parB 0.08
  y parB disk2read 2.1
  s parReply 0.01
  s loopOperation 0.1
  y loopOperation disk1read 0.7
  s loop2 0
  s bigLoopDriver 0
  f bigLoopDriver 1 # Exactly one call operation (deterministic)
  y bigLoopDriver bigLoop 1 # trigger the pseudo-task for the complex loop
  s seqReply 0.005
  s loopEnd 0
:
  serverStart -> (0.6)parInit + (0.4)seqInit;
  parInit -> parA & parB;
  parB & parA -> parReply;
  parReply[server];  # reply for the parallel branch.
  seqInit -> 3.5 * loopOperation, loopEnd;
  loopOperation -> loop2; # this activitiy is also in the loop.
  loopEnd -> 1.2 * bigLoopDriver, seqReply; # big loop is executed 1.2 times on ave
  seqReply[server] # reply for the sequential branch.
-1

A BigLoop
  s first 0.01
  s second 0
  y second disk1write 1
  s third 0
  y third disk2write 1
  s fourth 0.13
:
```

```
  first -> second & third;
  third & second -> fourth;
  fourth[bigLoop]
-1
```

Index