Layered Performance Modeling and Layered Queueing: Quick Tutorial

Murray Woodside, Carleton University
RADS Lab....http://sce.carleton.ca/rads
cmw@sce.carleton.ca
Mar 1, 2000

1.0 Basic concepts

Resources, authority, layering

Layered modeling describes a system by the sets of resources that are used by its operations. Every operation requires one or more resources, and the model defines a resource context and an architecture context for each operation. The architecture context is a software object to execute the operation, and the resource context is a set of software and hardware entities required by the operation.

Every resource includes an aspect of an authority to proceed and use it, which is controlled by a discipline and a queue (which may be explicit or implicit). In layered modeling the resources are ordered into layers (typically with user processes near the top and hardware at the bottom) to provide a structured order of requesting them. With layering a graph of all possible sequences of requests is acyclic, and deadlock among requests is impossible. For this and perhaps for other reasons, layered resources are very common in practice. Layering provides an order; requests may jump over layers.

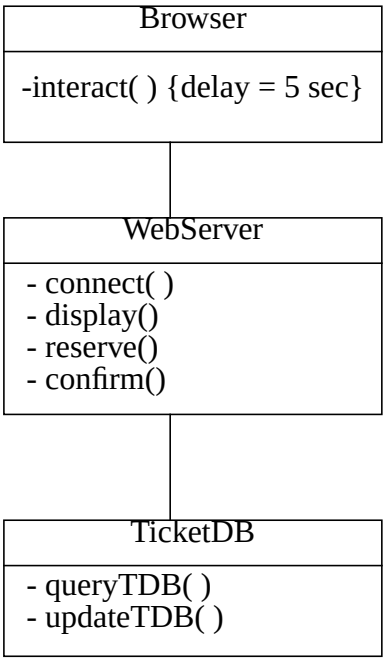
Tasks, entries, calls, demands

LQNS (Layered Queueing Network Solver) provides a language for modeling layered systems. It defines a system in terms of objects called *tasks,* that have resource properties. A task may model an object providing operations, a resource, or both together. A task offers one or more services through methods called *entries*, and these entries execute, and make *calls* to other entries. The objects in a model can be of any granularity, but usually they are large grained objects such as operating system processes. Figure 1 shows a diagram for a software system, using UML notation, and its LQN model.

The performance model describes the software by the average behaviour of the entries:

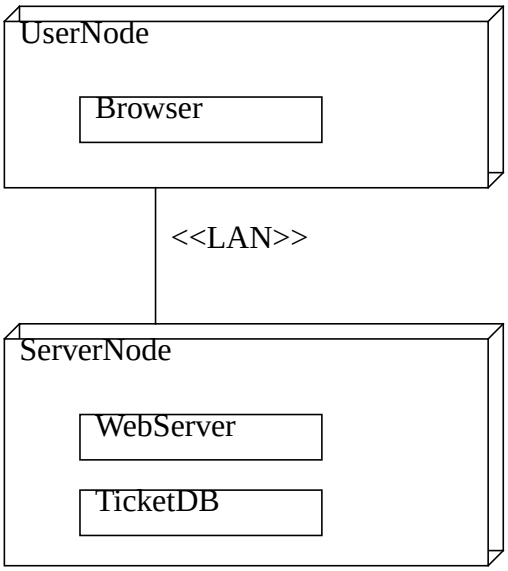
- execution demand, which is the average CPU demand for one invocation of the entry,
- demands for other resource services, stated as the average number of calls to each other entry, for one invocation of the entry,
- maybe a pure “wait” delay (stated as the average value for one invocation of the entry).

All operations are described by entries of “tasks”, even (for consistency) the operations of hardware devices. Thus a disk operation is described by a disk operation “task” running on a disk device, in lieu of a CPU. Every task has a host device or processor, which is normally a CPU node for a software object, and the processor may have a speed ratio

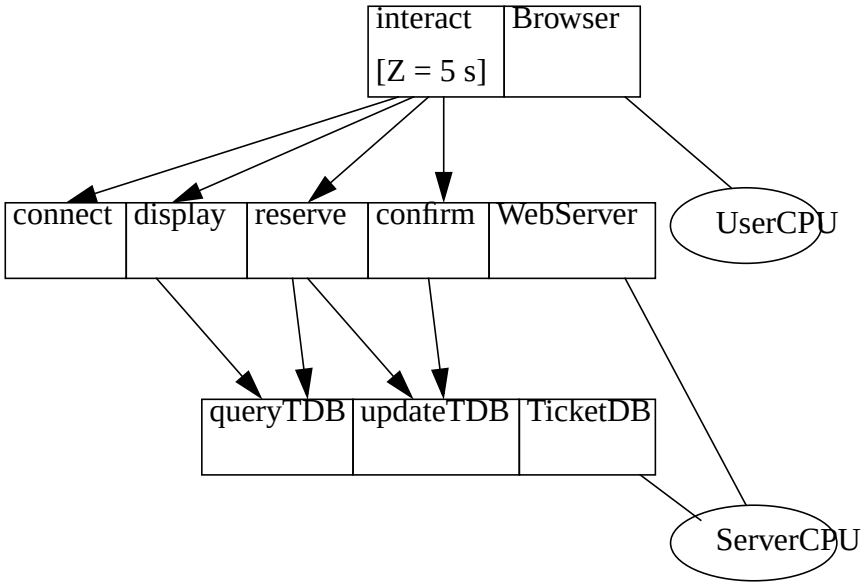


Layered system example of a web-based ticket reservation system:

UML class diagram

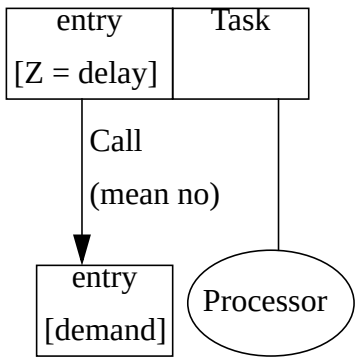


UML deployment diagram



Layered system example of a web-based ticket reservation system:

Layered Queueing model



Layered queueing notation

2.0 Task or process resources, queueing, and threads

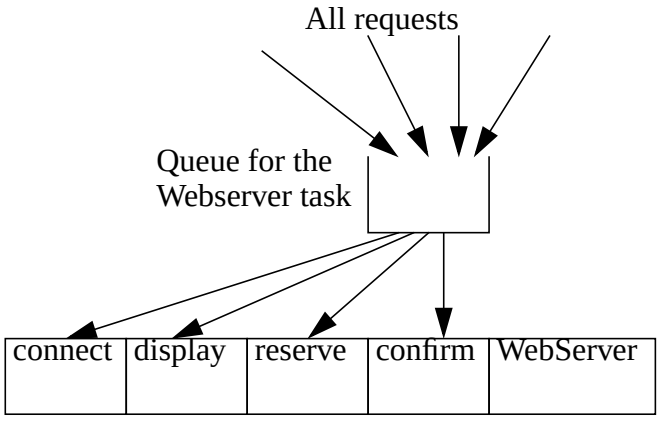
An object may exist in a single instance, such that only one request can be served at a time; this is called a *single-threaded task* and is identified with a *task resource*, and a task queue. One task queue is used by all requests to all the task's entries. This is an example of a "task" with both properties of an object providing operations, and of a resource.

Some software objects are fully re-entrant, they 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*. This is an example of a "task" which models an object providing operations, but imposes no resource constraint. Paradoxically an infinite resource is one which we do not have to consider as a resource at all, since it does not limit its authority to proceed.

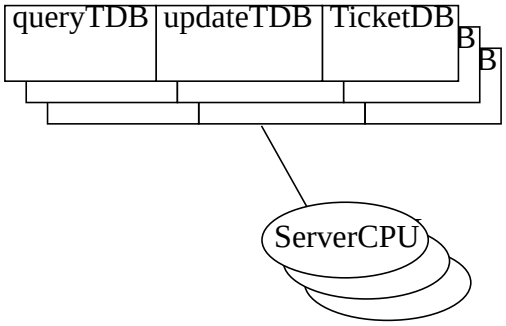
Some software objects exist as a pool of instances of a certain size. Requests are given a thread as long as there is one free; beyond this, requests must queue. These will be termed *multi-threaded.* Such a task models an object providing operations, and a homogeneous set of resource units that are dispatched from a single queue, like a multiserver.

Multiple instances may be provided in different ways, for instance by process forking, by lightweight threads or kernel threads, by re-entrant procedure calls, or by carefully writing a process that accepts all requests, and saves the context of uncompleted requests in data tables (we term this *data threading*).

Multiplicity also applies to processors (single, multiple, infinite).



A task has a single queue, for messages to all its entries



A multithreaded task running on a multiple processor

3.0 Blocking calls and other styles of interaction

Layered modeling in LQNS recognizes three kinds of interactions between entries:

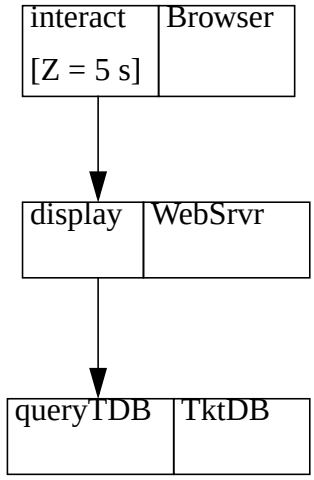
- synchronous call; the sender waits (blocked) for a reply. The receiving entry must be arranged to provide replies. This is the pattern of a standard remote procedure call (RPC). The sending object task resource or thread is kept busy during the wait. Software which does not wait is modeled as if it has a resource that does wait, perhaps one of an infinite pool of threads.
- asynchronous call: the sender does not wait and receives no reply. The receiving entry operates autonomously and handles the request.
- 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.

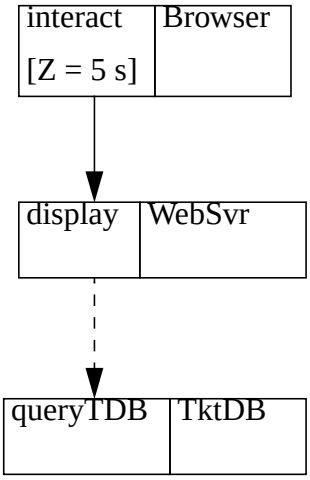
Additional styles of interaction may be constructed, including an asynchronous or delayed remote procedure call in which the sender continues at first, and eventually waits to get a reply.

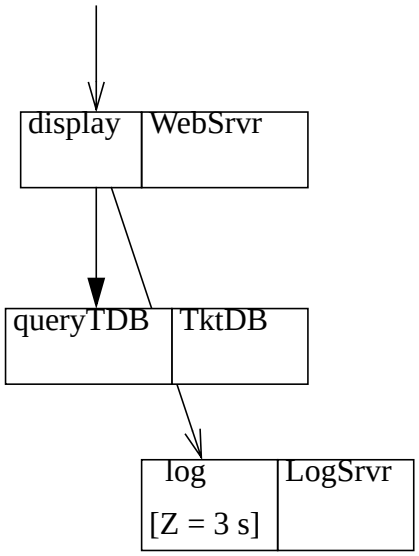
In each interaction there is a calling party, the “client” task, and a called party, the “server” task. A deeply layered system will have middle-level tasks that act both as servers, accepting calls from above, and as clients, making calls to lower-level servers. There may be tasks representing system users, that only originate requests. These pure client tasks act as sources of work, cycling between their own execution or delays, and requests into the system.

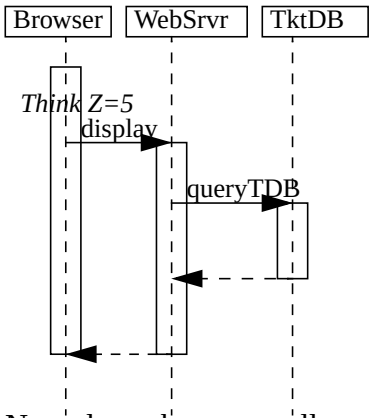
The system may also receive an arrival flow of requests from outside; these are treated as asynchronous requests from the environment (there is no reply).

For modeling purposes, such a request may activate an artificial infinite threaded “response” task, introduced into the model to capture the response delay. The response thread can be introduced so that it makes a synchronous request into the remainder of the system and waits for the completion of the activity.

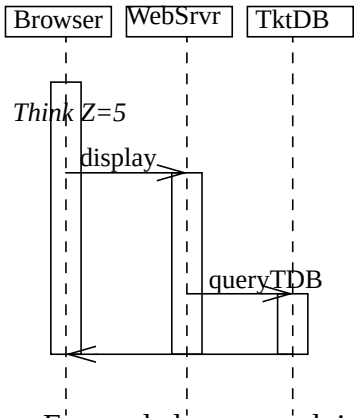




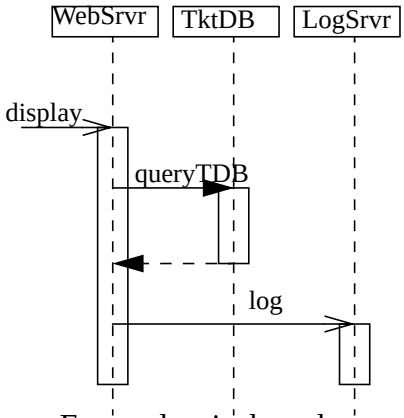




Nested synchronous calls: LQN above, interaction diagram below



Forwarded query, replying directly to the client



External arrivals and an asynchronous invocation

4.0 “Service time” of an entry or a task.

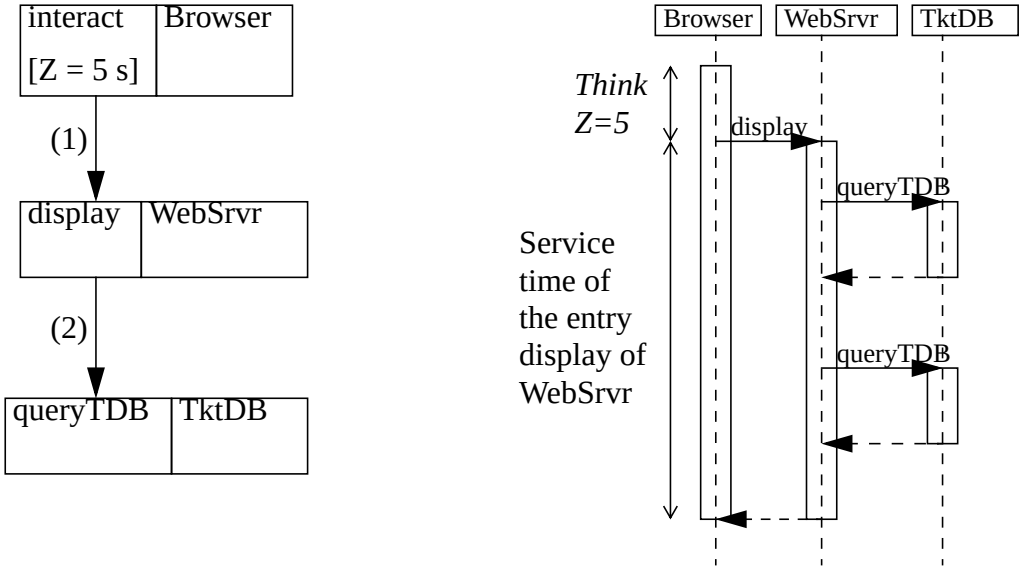
The service time of an entry is the time it is busy, in response to a single request. It includes its execution time and all the time it spends blocked, waiting for nested lower services to complete.

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

The utilization of a single-threaded task is the fraction of time the task is busy, meaning not idle (executing or blocked).

A multi-threaded task may have several services under way at once; its utilization is the mean number of busy threads.

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



The service time of an entry includes any nested service times while it is blocked, waiting for a reply

5.0 Mapping software activities into a layered model

If we begin from a view of the software as a distributed program executing a series of activities, we can describe it by one of many flavours of time-flow diagrams, such as task graphs, or execution graphs (Smith). Essentially such a graph defines the precedence relations between the activities. We shall use the notation

- activity1 -> activity2..... activity 1 precedes activity 2
- activity1 -> activity2 & activity3.....activity 1 precedes activities 2 and 3 (list of any length) (this is an AND-fork)
- activity1 & activity2 -> activity3..... (AND-join) (also may be a list of any length)

•

•

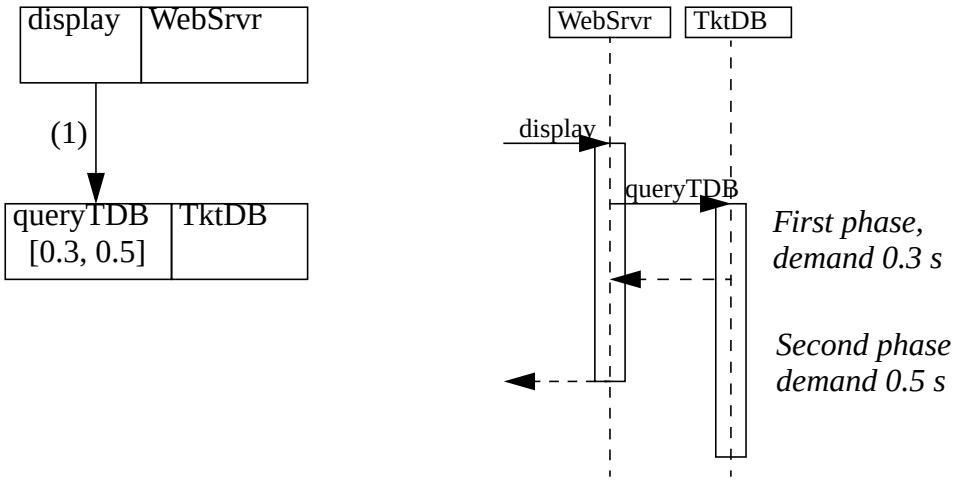
An activity can in principle be a large comp[lex operation distributed across several concurrent tasks, but for our purposes the description must be refined until each activity represents execution in a single operating system task, plus calls to other tasks.

It is convenient to combine the activity graph for flow of major activities with calls to services, and this was also reccommended by Smith in her execution graphs. For this purpose we may use the graphical notation shown in Figure XXX.

A call to a service maps into a call to an entry of a task. A transition between two activities that are in different tasks also maps into an intertask interaction, which can be recognized as synchronous if the flow returns to the same entry, asynchronous if it does not return or if it is also a fork in the flow, and forwarding if it returns by way of other tasks.

6.0 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 are done under sole control of the server task, and they form the second phase.



A second phase of service lets the client of the interaction proceed

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 congtrol 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.

The workload parameters originally described for an entry are given for all the phases of the entry. Each phase has

- mean total execution demand
- wait delay (also called a *think time*) (optional... it can be used to model any pure delay that does not occupy the processor)
- mean requests to each other entry

Additional optional phase parameters, not discussed before, are:

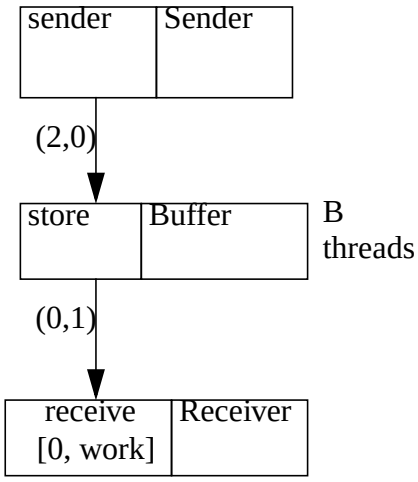
- the coefficient of variation of the execution demand requests
- a binary parameter to identify *stochastic phases* in which the number of nested requests is random, with a geometric distribution and the stated mean, from *deterministic phases* in which the number is exactly the stated number (which must be an integer).

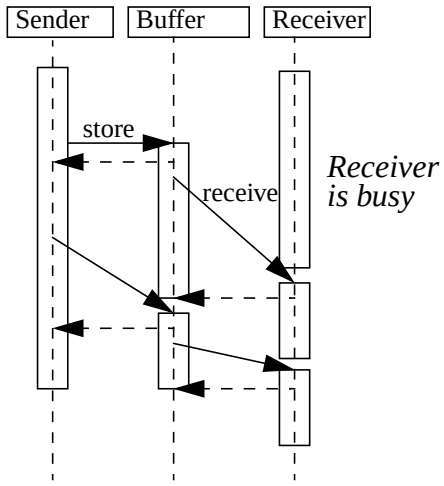
Second phases improve performance; they give less delay to the client, and they increase capacity because there can be some parallel operation between the client and the server. The amount of gain depends on circumstances (real parallelism needs separate processors, and a saturated server cannot increase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

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

figure of results

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. 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).





Modeling a finite buffer with blocking by a pool of B threads.
The buffer is full at the time of the second store.

7.0 Pipelines, and a third phase

In modeling software pipelines with acknowledged handovers, it is essential to separate the three operations of one task modeling a single pipeline stage... the input handshake (phase 1), the pipeline operation (phase 2), and the output handshake (phase 3).

The reason for separating them is technical; if the output handshake is averaged into phase 2 it will occur, on average, in the middle of the phase, and this will distort the pipeline delay!

This points out the need for a more general view of the structure of a “service”.

8.0 Activity sequences in an entry or task

The idea of phase can be made more powerful if we rename them as “activities” and allow any number of activities, with any kind of precedence relationship, including forking and joining of flows. The definition has two parts, a precedence part which defines a precedence graph, and a workload part. In the precedence graph, activities which trigger replies or forwarding of the request are identified. In the workload part, each activity has the workload parameters of a phase, as described above.

Each task has one precedence graph which covers all those entries which are defined by activities. Most often this graph is actually just a group of separate graphs, one for each entry, however a single graph could arise from two entries, and join their two flows together.

An entry can be defined *either* by phase parameters, as described in the earlier sections, or by identifiying the first activity in the entry. In the latter case the activity sequence definition completes the definition of the entry.