Performance Analysis of Distributed Server Systems

Greg Franks* Shikharesh Majumdar* John Neilson† Dorina Petriu*
Jerome Rolia* Murray Woodside*

* Department of Systems and Computer Engineering, Carleton University
† School of Computer Science, Carleton University

Abstract

It is generally accepted that performance characteristics, such as response time and throughput, are an integral part of the factors defining the quality of software products. The relationship between quality and system responsiveness is especially strong in the case of distributed application using various kind of software servers (name servers, application servers, database servers, etc.) In order to meet the performance requirements of such systems, the developers should be able to assess and understand the effect of various design decisions on system performance at an early stage, when changes can be made easily and effectively. Performance analysis should then continue throughout the whole life cycle, becoming one of the means of assuring the quality of software products. For this to become a practical reality, we need appropriate modeling techniques.

This paper presents a new performance model named *Layered Queueing Networks* (LQN) for systems with distributed software servers. In such systems the servers are are frequently layered, so that lower level delays are included in the higher layer service time, which limits the system capacity. LQN represents these effects in a compact and understandable way, and provide analytical and simulation tools. The paper explains the model, and more importantly, shows how can be applied to identify performance trouble spots in a system and devise effective corrective measures.

1 Introduction

System users are increasing their reliance on distributed applications (including client-server systems) to accomplish their business tasks. In such systems there is a strong relationship between their responsiveness and their quality. As is the case with mainframe environments, it is essential that the performance behaviour of these applications be well designed and managed. Predictive modeling tools are needed to help support performance oriented design and management. Such tools have been available for mainframe environments for almost two decades [1, 10]. Unfortunately, these tools are not able to characterize and model the behaviour of distributed applications. In this paper we present a new model for distributed applications. The application may be distributed across processors on a common bus, or across a wide area network, exploiting new midware technology such as DCE [4], CORBA [6] and Encina. The model is based on queueing network models and has relatively few parameters but is able to characterize many of the features that

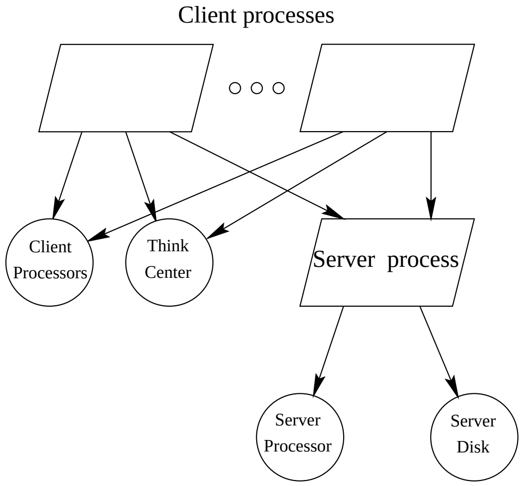


Figure 1: File server application

affect the behaviour of distributed applications. Analytic performance evaluation techniques have been developed for the model and are demonstrated with a sample application.

One of the most interesting performance aspects of distributed systems is that software processes can act as both clients and servers to other processes. Figure 1 illustrates several user processes sharing a file server process. The processes are shown as parallelograms, an arc shows requests for service and circles represent devices. Since the server process can have many client processes, it is possible for queueing delays to arise at the server process. Our model calculates the service time of a process, seen as a server, from its own execution and its requests to other processes, including queueing delays. The fact that this service time is not known *a priori* is a major difficulty in the design and understanding of software server systems. A server may become a software bottleneck, thus limiting the potential throughput of the system. This can occur even if the devices used by the process are not fully utilized. Workload characterization for distributed applications must capture these effects.

A *Layered Queueing Network* (LQN) model [5, 11, 13] is an extended queueing network model that also specifies visits between processes so that their *layered* requests for service and hence layered contention effects are represented. The parameters of the layered queueing network model are those of the queueing network model:

- customer classes, and their associated arrival rates or populations;
- devices, and their scheduling disciplines;
- for each process seen as a customer to a device:
  - the average number of visits to each device;
  - and the average service time at the device;

with additional parameters to capture software interactions:

- for each process seen as a customer to another process:
  - the average number of visits.

Predictive analytic modeling techniques based on approximate mean value analysis are then used to provide performance estimates for the behaviour of the system being studied.

There are many performance issues that can be considered using the model. These include studying the impact on response times of:

- using different algorithms for implementing a server's functionality;
- balancing loads across servers and/or changing device service rates;
- replicating (making copies) or threading a server;
- and having different numbers of customers.

The model and the evaluation technique are appropriate during the design of software as well as for system sizing, tuning, and capacity planning.

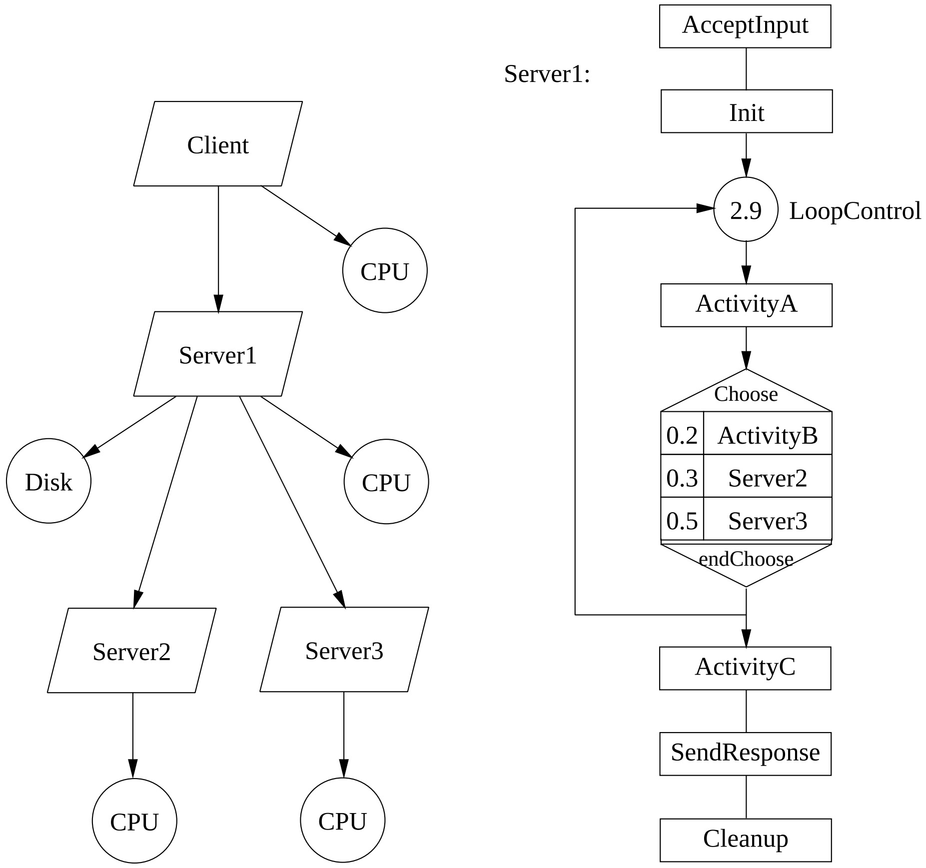
In section 2, software performance engineering techniques are described that can be used to characterize customer classes in layered queueing network models. Section 3 describes the features of LQN's. Sections 4 and 5 give an example of a distributed application architecture that is modeled as a LQN model and explore its performance behaviour using the analytic modeling techniques. A summary is given in Section 6.

2 Software Performance Model of a Server

To give informal insights into the layered model we will examine a typical server process in the configuration shown in Figure 2. The Client process executes repeatedly, periodically making a request to Server1, and waiting for the reply. While Server1 is carrying out its service, it makes use of two other processes called Server2 and Server3. We assume that these simply execute and return a result to Server1. Server1 will be considered in more detail, and model parameters will be derived from a behaviour description. For each request made to Server1, we want to know the total service time S for each device used by Server1 and the mean number of requests V made to each other server.

The analysis approach described by Smith [12] is adapted here to the case of layered software. It can be applied before the software is written. It uses an execution graph as shown in Figure 2(b) to describe the flow of operations followed in giving a service. The figure describes a generic sort of service, with no particular application in mind, but it illustrates the power of the analysis technique and the meaning of the model parameters. The initiation of the service begins at the top of the graph, with an activity which accepts the request, and follows downwards. The only difference from Smith's treatment is that we regard a request from Server1 for a service from Server2 or Server3 as a primitive resource request, that needs no further analysis. The number of these requests becomes the V parameter in the layered model, defined below.

The behaviour described in Figure 2 shows a generic service, which we shall assume is the only service offered by Server1. There is a sequence of activities (the boxes), a loop (indicated by the arrow and the circle showing the mean loop count) and inside the loop another sequence including a set of alternatives or cases, set off by triangles and including a probability for each case. (This notation is slightly different from Smith's for cases, but has the identical significance.) At the end of the service there is an activity for sending the reply to the requester, and a final "clean-up" activity for an operation which is performed after sending the reply but before accepting the next request.



(a) Server Configuration

(b) Execution Graph

Figure 2: A software server and its execution graph

Table 1 gives the activities in Figure 2(b) and shows the resource loadings. There is a line for each activity, and lines for the control overheads for the loop and for the cases. There is a column for each resource used by the activities, and resources can be physical or logical. A reduction process is described below which determines the total loadings to physical resources and software servers.

The use of a physical or logical resource is measured in terms of its operations. In the case of the CPU, a unit of a million operations (MOp) has been used. The use of Server2 or the disk (at a logical level) is measured in terms of requests, called a "Server2 request", meaning a request to Server2 sent over the network, or a "disk access", meaning a logical, program-level operation on a disk file. One program-level file operation may lead to several physical disk operations, depending on the amount of information being read or written, or may lead to none, if there is an effective disk cache. Therefore we first define the logical operations which we see in the software design, and then convert to physical operations.

To obtain a performance model the table is reduced in four steps:

Activities	Repetitions	Resource Loadings (per repetition)				
		CPU (MOp/S)	Disk accesses	Server2 requests	Server3 requests	IPC rqsts.
AcceptInput	1	0.3				1
Init	1	0.7	3			
LoopControl(2.9)	3.9	0.05				
ActivityA	2.9	6.5	2			
Choose	2.9	0.02				
.2 ActivityB	0.58	12.0	11			
.3 Server2	0.87			1		
.4 Server3	1.16				1	
endChoose						
ActivityC	1	4	1.5			
SendResponse	1	0.2				1
Cleanup	1	2.4	5			

Table 1: Performance Parameters of the Execution Graph in Figure 2(b)

- The logical operation requests are reduced to physical demands by substituting estimates of the physical operations per logical operation. For example, a logical disk operation may give rise to sever disk device operations and some CPU time. In this reduction, requests to other software servers are treated as physical demands since they are not resolvable by the physical resources used directly by this server process.
- The mean repetitions are multiplied through each row, to give total demands for that activity.
- The physical operation counts are reduced to time estimates, by substituting the expected time per operation (e.g., the operation time for the CPU). The service requests are untouched at this step.
- The time estimates for each resource are added up to give a total demand for one invocation of the service. Two sums are formed for each resource, one sum for the activities between receiving the request and sending the response, called "phase 1", and one sum for the activities after that, called "phase 2". The service requests to other servers are also summed, once for phase 1 and once for phase 2.

In this way, for each of phase 1 and phase 2 we obtain a set of immediate resource demands by this server (Server1) for the devices it uses, in average seconds of work per input request, and the average number of requests to other servers. We can write these as:

- S(server, device, phase) seconds, for "server", using "device", in phase "phase";
- V(server, OtherServer, phase) requests, for "server" requesting to "OtherServer", in phase "phase".

In the example above, for instance, V(Server1, Server2, 1) = 0.87 since on average there will be 0.87 requests to Server2 per invocation of this service, and the requests are in phase 1, before sending the response. S(Server1, CPU, 2) is the CPU time demand of the Cleanup activity, which (from the Figure) is the time demand of 2.4 million operations plus the CPU demand required for 5 disk accesses.

The layered model derived this way includes a number of assumptions which may not be perfectly satisfied by the software. The system performance is assumed to depend only on the average execution time values on devices, and the average number of requests made to other servers by each process.

3 Layered Queueing Concepts

Layered Queueing Networks can be used to model a variety of different types of system behavior and inter-process communication styles. A standard RPC can be modeled by a server with a zero second phase of service. The send-receive-reply cycle of inter-process communication supported by the V [3] and Amoeba [8] operating systems is also modeled by an LQN. For certain classes of systems the second phase of service can be aptly used to model housekeeping operations or operating system overheads associated with the service that was provided in the first phase.

Distinguishing features of the layered model are:

- *diversity in modeling*: the processes in a LQN can be used to model hardware devices such a processors, disks, and communication links as well as software processes.
- *nested service*: a server during its response to a client request may request service from another server processes. During this nested service both the client and the first server are blocked. This models simultaneous resource possession which is difficult to handle by conventional techniques.
- *entries*: an "entry" identifies a particular service offered by a process, when the process offers more than one type of service. Entries are associated with server processes to handle these situations. A database server, for example, may be represented by a process with two entries: one that corresponds to a query operation and the other for an update operation. The performance parameters may be different for different entries.
- *Multithreading*: Multi-threading may be an important way to remove software bottlenecks, and will be shown in section 4. A server process, with multiple identical threads is modeled as a multi-server in a LQN.

4 LQN Model of a Transaction Processing System

This section presents a LQN model of a transaction processing system based on a distributed relational database, NonStopSQL, running on a Tandem multiprocessor system (see Figure 3). The hardware is a "share-nothing" architecture designed for scalability that minimizes the contention for common resources. The processors are interconnected by dual high-speed buses, and each processor has its private memory and I/O buses. The NonStopSQL software integrates the SQL functions with operating system functions in order to achieve a better performance [2]. The software architecture is based on concurrent processes communicating through messages in a client-server fashion. Because the processors are linked only by a communications subnet, and the data is

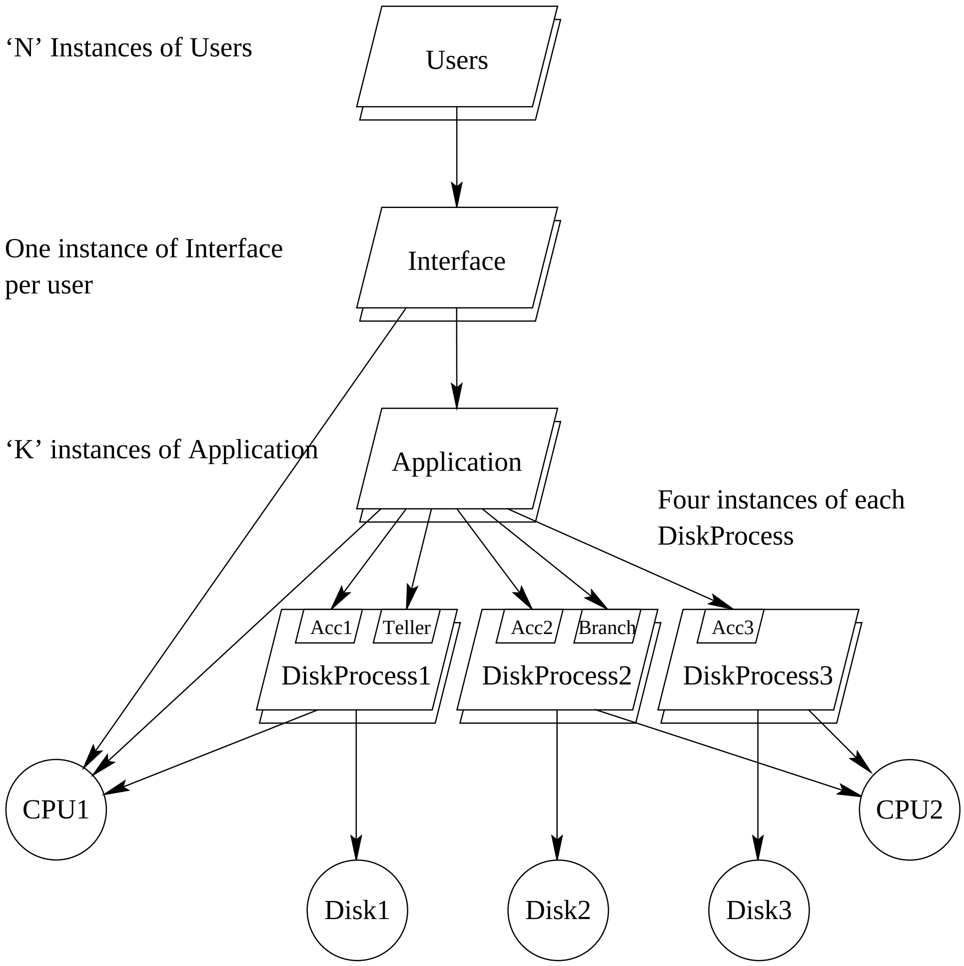


Figure 3: Distributed Database Example System

partitioned and distributed, the model has all of the features of a distributed system; only the communications delays are relatively short.

The workload considered in this example is a simplification of the Debit-Credit transaction described in [7]. The Users, represented by the processes at the top of the graph, think for a given time, then send their queries to an Interface process (one for each user) which manages terminal I/O, validates the input data, translates the queries, and routes the requests to the application process.

The application process decomposes a query into simple SQL operations, sends these to another software server called the DiskProcess for execution, then assembles the final response and replies to the user. More than one instance of the Application process can be active at the same time, processing different user queries from a common queue and running on the same processor. Therefore, we represent the Application process in the LQN model as a multi-server with a finite number of server instances.

A Disk Process is a software server that manages a disk unit. It integrates several important operating system and database management functions to improve performance [2]. For example, it implements simple SQL operations, performs logical and physical disk I/O, and manages the disk cache. More than one instance of a disk process can be active simultaneously on the same CPU managing the same disk unit and serving requests from a common queue. Thus, a disk process is represented in the LQN model as a multi-server with a finite number of servers, each one able to offer a range of services using a number of entries. All instances of the disk process are competing for the same hardware servers (i.e., CPU and disk).

The system supports data distribution with local autonomy thus increasing the potential for parallelism: both data and index files can be partitioned horizontally across several disks. We consider a database consisting of three SQL tables with vastly different sizes:

ACCOUNT: a table of bank accounts containing hundreds of thousands of records; each record holds the account number (also the primary key) and the balance.

TELLER: a table describing bank tellers, containing hundreds of records.

BRANCH: a table describing the cash positions of each bank branch, containing tens of records.

Our simplified Debit-Credit transaction reads the input data from the terminal, then updates the three tables in sequence. We did not model here the logging activity for simplicity reasons. This does not change significantly the workload in this model, since the logging is handled by a separate subsystem with its own CPU and disks.

We consider that the ACCOUNT file (by far the largest) and its primary index file are each partitioned into three fragments allocated on Disk1, Disk2 and Disk3, respectively (represented by circles at the bottom of the figure). The file TELLER and its primary index are allocated on Disk1, and BRANCH and its primary index on Disk2. In terms of software servers, the model contains three DiskProcess multi-servers (one for each disk unit), an Application multi-server and an Interface infinite server. Due to the way the files are allocated on disks, DiskProcess1 offers two services (*i.e.* entries) with different workload requirements: one to update the ACCOUNT fragment, the other to update the BRANCH file. DiskProcess2 is in a similar situation.

The software components are allocated on two processors: the Interface, Application and DiskProcess1 on CPU1, and DiskProcess2 and DiskProcess3 on CPU2 (see Figure 3).

5 Experimental Results

In this section the transaction processing model described in Section 4 will be used to illustrate how capacity can be estimated and the performance effects caused by varying degrees of server multithreading and disk caching.

Since the model is closed in the sense that it has a fixed number, N, of Users, we will estimate the mean response time for a number of N values. Each user is modeled with a simple operational cycle involving a mean "think" time of 10 seconds preceding each data base request. Other model parameters were estimated in the manner described in Section 2. Disk service request rates were estimated taking cache hit ratios into account.

The response time characteristics, as computed by the analytic LQN solver described in [5], are shown in Figure 4 for several values of the degree of multithreadedness of the Application Server, parameter K. Each curve exhibits the familiar nearly constant response time for light loads with a relatively sharp transition to a steeply sloping relationship when the loading is high. The position of the "knee" of these curves is often taken as a measure of the capacity of the system. For example, if mean response times of the order of one second were considered to be satisfactory for this system, system capacity in terms of the number of users would be approximately 225 users for the case of K = 16. This translates to a transaction processing rate of approximately 20.41 transactions per second.

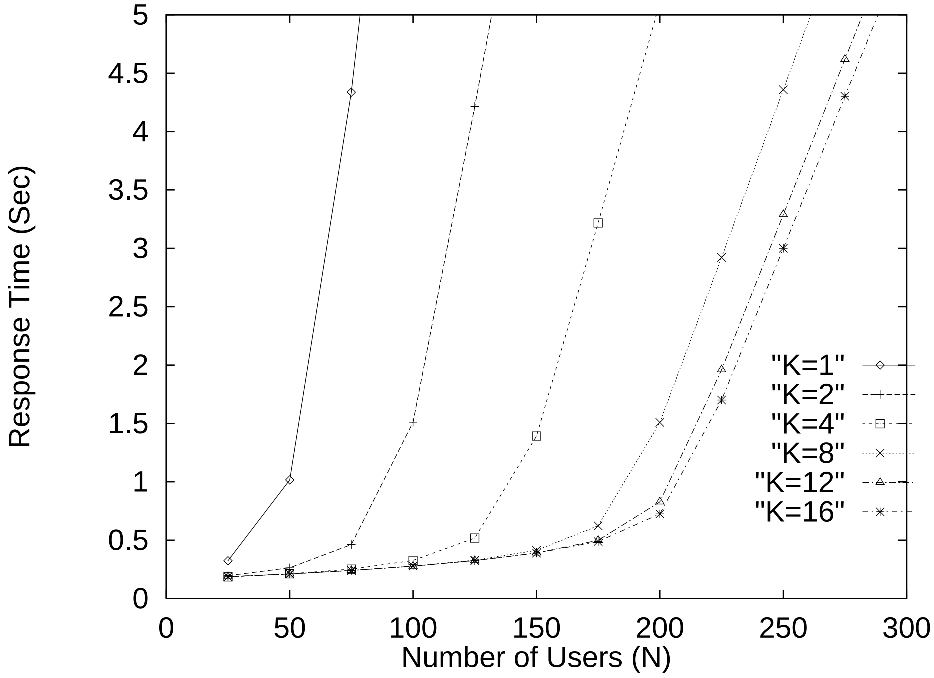


Figure 4: Response Time Characteristics

Transitions in response time curves are associated with the saturation of one or more resources when its utilization approaches unity. This is often referred to as bottlenecking. The universal remedy is to increase the capacity of the limiting resource or server and thereby reduce its utilization. With systems such as the one modeled, one must not to ignore the possibility that a limiting resource may be an intermediate or "software" server. Notice that such servers are "busy" in the sense that they are occupied and unable to handle subsequent requests even while they await the completion of service from subordinate servers. High utilizations of software servers may result with significant embedded service, i.e., when service is supplied by sub-ordinate servers while the server in question is blocked awaiting results. In part, this effect arises from the synchronous nature of the interprocess communication used here and prevalent in systems using RPC's.

To increase the capacity of a software server we can increase its number of threads. This effect is clearly demonstrated in Figure 4. Consider the case when K = 1, i.e., there is a single instance of the application server. Performance is poor with the response time knee well to the left. As the capacity of the application resource is increased by raising K, the performance is greatly improved. Note that this is accomplished without adding to the capacity of any other resource including the processor and disk devices. What is witnessed here, for small K values, is a phenomenon known as software bottlenecking [9]. Our example demonstrates that its effect can be very significant —

more than a factor of 4 in system capacity from K = 1 to K = 16.

To understand the software bottlenecking phenomenon better, consider Figure 5. Shown are the utilizations of several of the servers, of both the hardware and software types, as functions of the parameter K. When K is small, say $K < 4$, the utilization of each instance of the Application Server is essentially unity. All other servers are comparatively lightly loaded with utilizations less than 0.7. For larger values of K, Application Server utilizations fall, eventually below those of CPU1 — the processor on which instances of the Application Server execute, as well as below those of DiskProcess1, another "software" server. Note that the cross over point between the utilizations of the Application Server instances and CPU1 occurs at a K value of approximately 11. For larger K values, it is the hardware server CPU1 that dominates in limiting performance. This explains the small performance gain when K is increased from 12 to 16 in Figure 4.

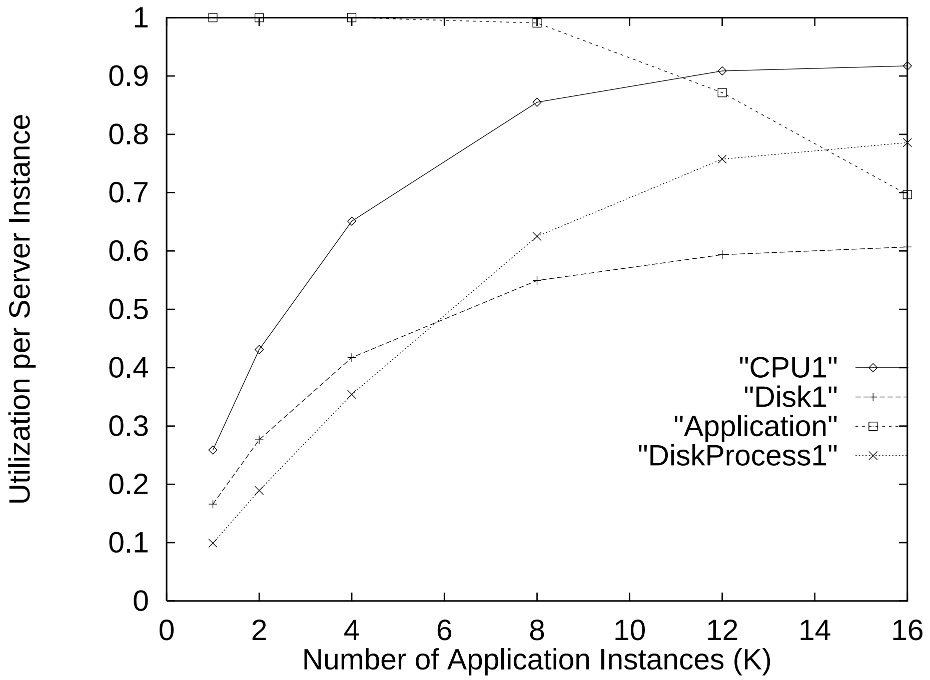


Figure 5: Selected Server Utilizations verus K

One may ask: Why limit the degree of multithreading of software servers? One answer is the cost in terms of memory as each thread requires an execution stack that is often significant in size. Also, threads typically require synchronization mechanisms, e.g., locks, mutexes, or semaphores for use with shared data. These can limit the "effective" degree of multithreading. Thus, a sensitivity check on the degree of multithreading is useful to guide design.

Use of LQN models to provide sensitivity analysis is particularly important in the early stages of design when parameters are not well known. For example, in the transaction processing model, disk caching is used to improve performance. Estimates of cache hit ratios were necessary to provide disk service request parameters for the DiskProcesses. If these cache hit ratio estimates were questionable, the sensitivity of the system performance to cache hit ratios would be useful to know. Figure 6 shows the effect, on the "base" case response time characteristic, as cache hit ratios are varied. "Base" case cache hit ratios ranged from 0.72, for the Teller File, to 0.9, for the Branch file. In this study, disk cache misses were increased by 50% for the "low" cache hit cases and halved for the "high" cache hit cases. Notice that slightly dropping the disk cache hit ratios from their relatively high values had a very significant effect on shifting the response "knee". The difference between the low and the high hit ratios is worth about 50 users in system capacity!

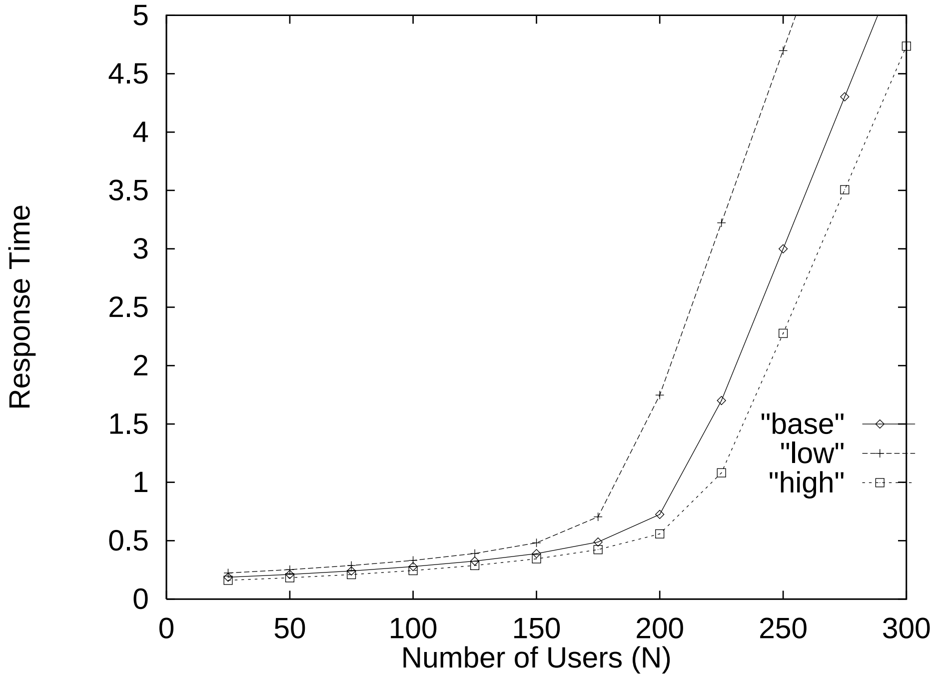


Figure 6: Disk Cache Hit Ratio Sensitivity Study

These experiments illustrate the power of a LQN performance model in helping to answer design questions. Often these questions are associated with resource allocation but performance-resource relationships are non-trivial. The analysis given for this example could be used to best apportion memory between thread stacks and disk caches.

6 Summary

This paper has introduced a performance model which represents software resources and queueing of requests for software servers explicitly. It can be used throughout different life cycle stages, from early design to maintenance, to insure that performance is built into the systems and is effectively managed. The model captures the important features of nested service at subordinate servers. The model parameters can be derived by conventional methods as shown in Section 2. A transaction processing example has illustrated the use of multithreading to increase performance.

The analytic model solver has been found to be fast, typically more than an order of magnitude faster than comparably accurate simulations, and sufficiently accurate for planning and design purposes.

References

[1] BGS Systems, Inc., Waltham, MA. *Best/1 User's Guide*, 1982.

[2] Andrea Borr and Franco Putzolu. High performance SQL through low-level system integration. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 342–349, Chicago, IL, June 1988.

[3] D. R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[4] Open Software Foundation. *Introduction to OSF DCE.* Prentice Hall, first edition, 1992.

[5] Greg Franks, Alex Hubbard, Shikharesh Majumdar, Dorina Petriu, Jerome Rolia, and Murray Woodside. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation*, 24(1–2):117–135, November 1995.

[6] Object Management Group and Xopen. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open, Framingham, MA and Reading Berkshire, UK, 1992.

[7] The Tandem Performance Group. A benchmark of NonStop SQL on the debit credit transaction. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 337–341, Chicago, IL, June 1988.

[8] Sape Mullender, Guido von Rossum, Andrew Tanenbaum, Robbert von Renesse, and Hans von Staveren. Amoeba: A distributed operating system for the 1990's. *IEEE Computer*, 23(5), May 1990.

[9] John E. Neilson, C. Murray Woodside, Dorina C. Petriu, and Shikharesh Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(9):776–782, September 1995.

[10] Quantitative System Performance, Inc., Seattle, WA. *MAP User's Guide.*, 1982.

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

[12] Connie U. Smith. *Performance Engineering of Software Systems.* The SEI Series in Software Engineering. Addison-Wesley, 1990.

[13] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(8):20–34, August 1995.