Performance Engineering of Client-Server Systems

C. Murray Woodside
Real-Time and Distributed Systems Group
Carleton University
Ottawa K1S 5B6, Canada
email: cmw@sce.carleton.ca

Abstract

A great many distributed applications have a client-server architecture, and many more are being planned, including transaction processing systems, network services, and computational services. Almost all systems based on remote procedure calls (RPC) have a client-server structure and distributed processing architecture such as DCE (Distributed Computing Environment) from the open System Foundation do too. The performance of these systems has an unsettling characteristic because of the blocking nature of almost all RPC, that the service times of the software server tasks are not constant, but increase with load. This makes it more necessary than normal to model the performance of the system, using a model that incorporates this effect. Further, this effect also can modify the location of bottlenecks, and create "software bottlenecks" which are difficult to diagnose in advance.

1.0 Introduction

Client-server architectures are suitable for a wide variety of distributed systems. For this reason several proposals for standards for open distributed computing are based on a client-server architectures, with remote procedure calls (RPC) to servers. The DCE "Distributed Computing Environment" of the Open System Foundation is an example [1]. Also, operating systems such as SUN-OS, Mach and V which support distribution often provide many standard services through RPC to servers outside the kernel. Because of the many different potential sources of delay (in communications delays and overheads, in the server themselves, and in queues of requests at congested servers), performance must be carefully engineered.

This tutorial describes the performance aspects of client server (CS) systems. It begins with a canonical CS notation describing the relationship of the various processes, with examples. The notation includes parameters which summarize the workload in a simple way. A feature of the CS notation is that it treats resources of all kinds - software and hardware - uniformly. Once a system has been described with structure and parameters it can be solved by simulation, by well-established techniques, or by analytic calculations. The basis of the analytic methods is described.

A special phenomenon in CS systems is the *software bottleneck*, in which a software process can be effectively saturated by time it spends blocked waiting for responses from other servers. This may cause a system to saturate even though its processors are substantially under-utilized. The phenomenon of software bottlenecks is explained and their removal by *cloning* is studied in the final section.

2.0 The CS Notation for Client-Server Systems

The basic relationships in a Client Server system are illustrated in Figure 1. Requests are indicated by an arrow from the client (which sends the request in a message) to the server (which serves it and sends a reply). A "pure client" task only sends requests, as at the top

of the figure; a pure server only receives them, and some tasks like the one in the middle do both.

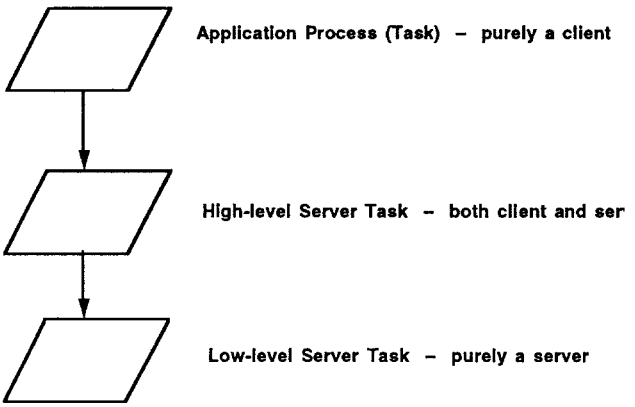


Figure 1. Clients and Servers: Architectural Relationships

The oval indicates a processor used by the middle server; in the CS notation a processor is also regarded as a server to the software task. For simplicity the processor servers may be omitted in some diagrams. Figure 2 shows the parts of a generic distributed database system. Figure 3 shows a frequently-discussed client-server configuration, in which the users and their terminals are also represented by "tasks" for the purposes of modelling, as are the disk controller and disk devices. The Client task typically does presentation management (screens and some verification) and the Server does the rest.

Notice that the CS architectural model is a *graph* with nodes for tasks and arcs for requests. The model assumes there are *no cycles* in the graph - that is, no closed path formed by following the arrows. Then the model has a set of topmost clients, representing human users or autonomous "demon" processes that generate work, and some pure servers at the bottom representing devices and processors.

In CS notation the service given by a server task may have two parts called "phases" as indicated in Figure 4. *Phase 1* is that part executed between receiving the request and sending the reply. For server i, phase 1 can be described by the parameters

s_{i1} = execution time on the processor for server i

y_{ij1} = the mean number of requests to another server j.

After sending the reply, *phase 2* is what is executed until the next message is received. It may be just necessary overhead, including buffer cleanup and task re-initialization or control code and in some cases a context switch (if there is no message waiting to be processed immediately). It may also include substantial work which is deliberately kept until after the reply to avoid delaying the client. A typical example is delayed writes to disk to

update data, or to log the transaction. The second phase has the parameters s_{i2} and y_{ij2}, corresponding to those for phase 1. This model is defined more fully in [2] and [3].

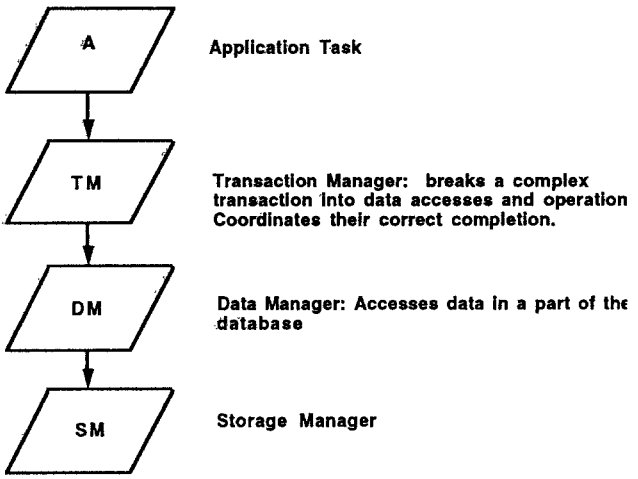


Figure 2. Example: On-Line Transaction Processing and Distributed Database Generic Architecture

The "service time" of a software server, as seen by its clients, is the total length of its first phase. The second phase cannot however be ignored, because it can delay the next client in the queue. The delay effect must be found by solving a performance model, for it depends on the probability that the next client arrives before the second phase is finished.

A Behavioral Model

The structural CS model with its parameters s and y (as defined above) is shown in its basic form in Figure 5(a). Figure 5(b) shows a flow graph for the CS view of the execution of one phase of task i, labelled phase p. In this view the execution is divided into "slices" of equal average length and the choice of whether to end the phase or to make a nested service request to another server is random. Defining $Y = 1 + \Sigma_j y_{ijp}$,

- The mean slice length is s_{ip}/Y
- the probability of ending the the slice with a request to server j is y_{ijp}/Y,
- the probability of ending the phase is $1/Y$.

The random choice models a data-dependent choice. Other behavioral models are possible. A particularly useful one is the "deterministic choice" model with exactly y_{ijp} requests to server j.

The random choice models a data-dependent choice. Other behavioral models are possible. A particularly useful one is the "deterministic choice" model with exactly y_{ijp} requests to server j.

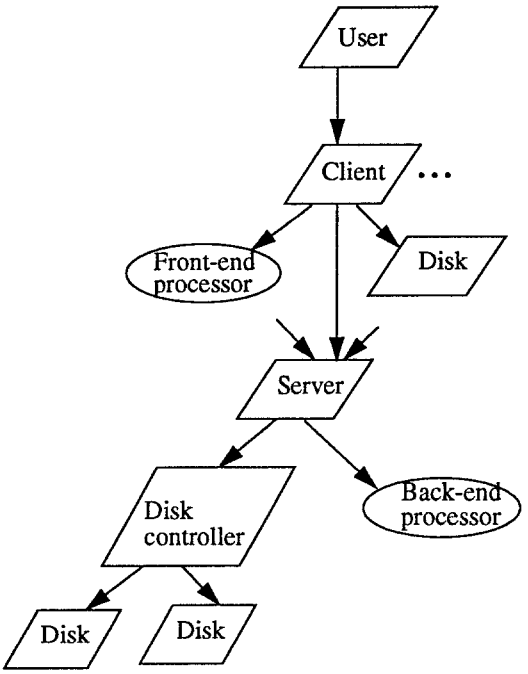


Figure 3. Client Front-end and Server Back-end

The topmost clients do not receive requests. In the CS model these clients *constantly cycle*. The cycle may include an average "think and type" time representing time before a user enters the next request. Seen as a "task" therefore the topmost clients are always busy. Since they never respond to requests or send replies their execution is modelled as *phase 2 only.*

3.0 Performance Factors in Client-Server Systems.

The service time of a software server is strongly load-dependent. This is the key property which makes client-server systems difficult to understand, and which makes performance modelling pay off. Figure 6 shows why. The service time x_{i1} is made up of execution intervals (shaded) alternating with gaps for nested requests to another server, say server j.

Both the gaps and the execution intervals, when closely examined, include a waiting time which is load-dependent, and a service time. If the lower level queueing delays increase, the server i service time increases, even though much of this time is just "gaps". In this way server i could be saturated, meaning it is never idle, even though it is not executing for most of the time. This is a "software bottleneck"

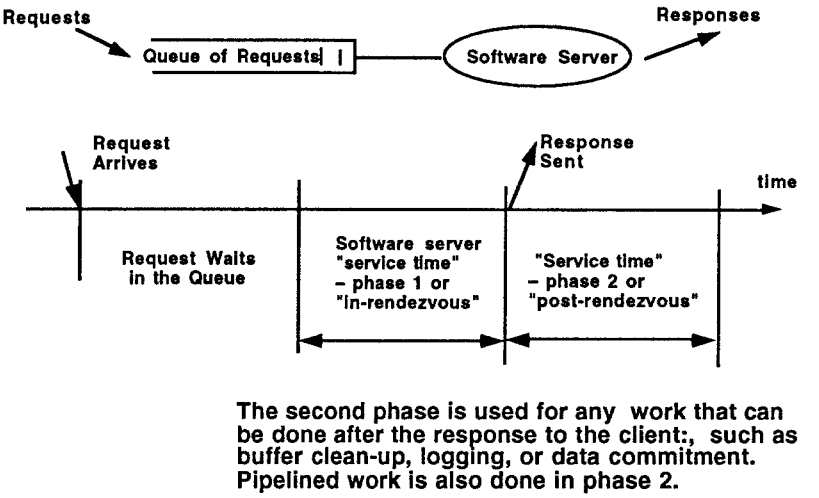


Figure 4. Synchronous Client-Server Interaction - A Template

Software bottlenecks tend to propagate upwards because they cause delays in higher level tasks in the CS diagram. This is illustrated in Figure 7. The left side shows a client task which makes infrequent requests from lightly loaded servers, which are therefore not very busy. The topmost client tasks in the CS model however are assumed to loop for ever and to never be "idle". This means that by convention the topmost client is always saturated, and it is shown shaded to indicate this. When only the topmost clients are saturated, as on the left, the system is lightly loaded.

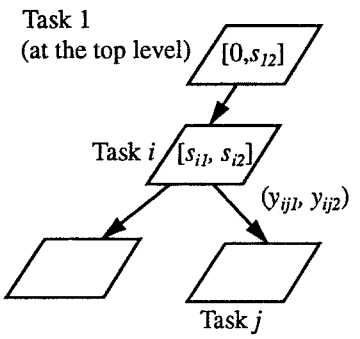
The middle and right-hand sides of Figure 7 show saturated servers. In the middle, the 2nd level is heavily loaded, perhaps by having more clients. This has a push-up effect and the client cycle times are now longer, due to waiting. On the right side, the bottom server is saturated by requests from many clients. It now has long waiting, which saturates the middle server even if, by itself, it would not be saturated; this then affects the topmost client response time and throughput.

The system *capacity* is the throughput (or set of throughputs) which saturates the system. Typically saturation has a focus at a *bottleneck point* like the middle server in the middle case of Figure 7, which in turn saturates tasks above it in the CS architecture. Because service times are load dependent, they must be found by analyzing contention - they cannot be found from an examination of static parameters in the model.

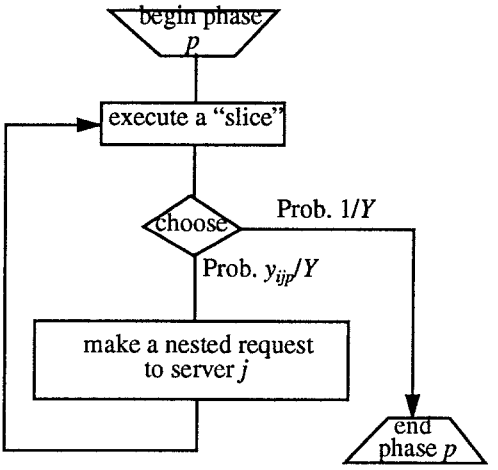
4.0 Solving for Performance in Client-Server Systems

The previous section showed that an estimate of the mean waiting time at some servers is necessary in order to compute the system capacity. Simulation can be used to get these estimates, or an analytic model can be used. This section shows some relation ships between quantities in the system, and briefly describes analytic modelling. It also

describes a slightly more elaborate version of the CS architecture including distinct task *entries*. The solution without entries is given in [2]; with entries, in [3].



(a) Structural Model



(b) Behavioral Model with Random Choice, for task i, phase p.

Figure 5. Structural and Behavioral Model of a Server

Components of Delay at a Server

The service time of server i is determined as follows:

- x_{i1} = phase 1 service time, between "begin service", and "send reply"
- x_{i2} = phase 2 service time, between send reply and next "receive request"

Supposing the server has a private processor, so it does not wait to execute:

$x_{ip} = s_{ip} + \Sigma_j y_{ijp} (w_{ij} + x_{j1})$

s_{ip} = execution time of server i in phase p

y_{ijp} = mean number of requests to server j in phase p

w_{ij} = mean queue wait at server j

x_{j1} = phase 1 service time at the nested server j

To include processor contention, let the processor also be a server. The parameters for requests from server i to its own processor are:

$y_{i,processor,p} = 1 + \Sigma_j y_{ijp}$

$x_{processor,p} = s_{processor,p} = s_{ip} / y_{i,processor,p}$

Then the service time of phase p is:

$x_{jp} = \Sigma_j y_{ijp} (w_{ij} + x_{j1})$ (where the sum over j includes the processor)

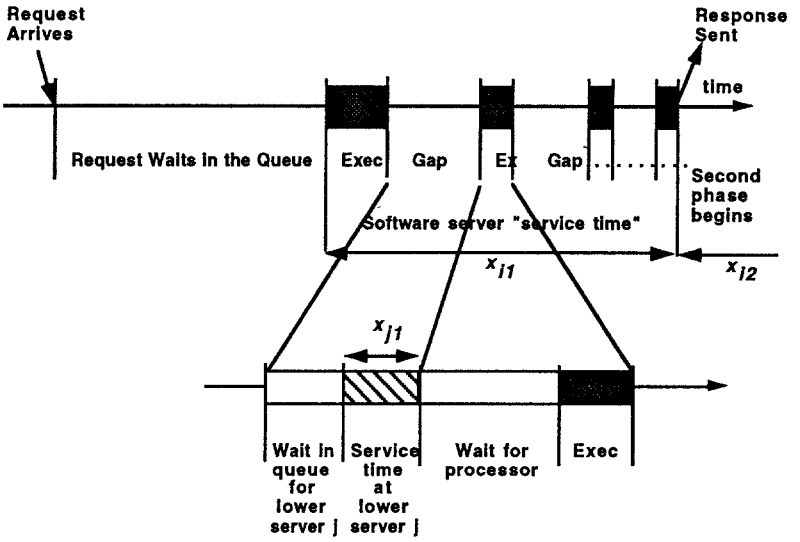


Figure 6. Recursive Expansion of Delays at Server i

Modifications for Separate Services ("Entries")

A server may offer several services, each with a separate address, or port, or (in a term used in the Ada language) "entry". We assume a single common queue with a FIFO discipline, or a priority queue in which priority is given to different entries.

Examples of different services are: *create record, read record, update record.* Figure 8 shows the modification made to the CS structure diagram for entries. The task node is divided and separate s and y parameters are attached to each entry. Figure 8 also shows an entry introduced to represent the processor execution of entry e, along the lines just described. The resulting equations are:

x_{ep} = service time of entry e, phase p. The entry belongs to task i.

$x_{ep} = \Sigma_d y_{edp} (w_{ed} + x_{d1})$

y_{edp} = mean number of requests to entry d, in phase p

w_{ed} = mean queue wait at server j, entry d

x_{d1} = phase 1 service time at the nested server j, entry d

As before, y and x for the processor of server j are found from s_{ep}.

To solve for the performance of a CS model:

- the *inputs* are the s and y parameters

the *outputs* are the w and x parameters, particularly the x_{i2} values for the topmost clients. These clients then have throughputs $1/x_{i2}$, for client i.

The only difficulty is in estimating w_{ab} between entries a and b.

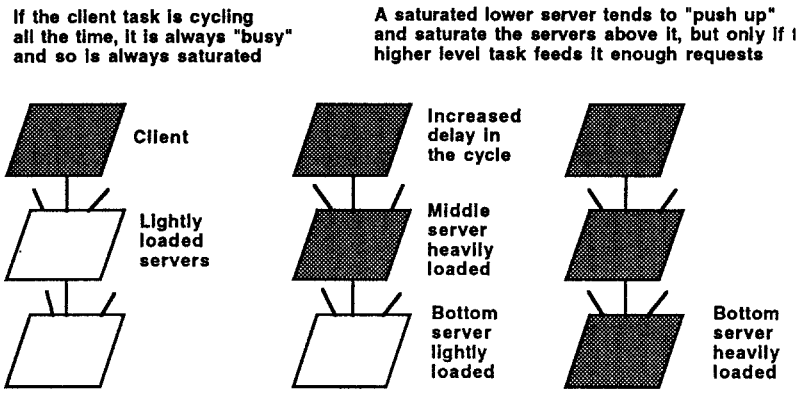


Figure 7. Vertical Dependency of Bottlenecks

Solving for waiting: Heuristic MVA [2]

An heuristic "mean value analysis" method has been developed. It finds w_{ab} as a sum of contributions from other tasks requesting service at b. Suppose a request from entry a of task i arrives to entry b of task j. It may find the server busy in various states, and it may find various competitor requests queued ahead of it. Its mean wait before starting service is:

$$w_{ab} = \Sigma_{c,d,p}\, V(a,b,d,p)\ \text{Prob}\{InService(a,b,c,d,p)\} + \Sigma_{c,d}\, x_{dp}\ \text{Prob}\{InQueue(a,b,c,d)\}$$

InService(a,b,c,d,p) signifies that the server is executing a request from entry c to entry d, and the server is in phase p of entry d. Then *V(a,b,d,p)* is the "mean residual time" for the entry d.

InQueue(a,b,c,d) signifies that a request from entry c to entry d is in the queue ahead of it, when a request from entry a of task i arrives to entry b of task j.

Figure 9 illustrates the arrival states.

The notation *V(a,b,d,p)* in the first term stands for the mean residual time of the service being executed at the arrival instant, if any.

Mean Residual Time *V(a,b,d,p)* = (mean residual time of entry d, phase p) + (phases beyond p)

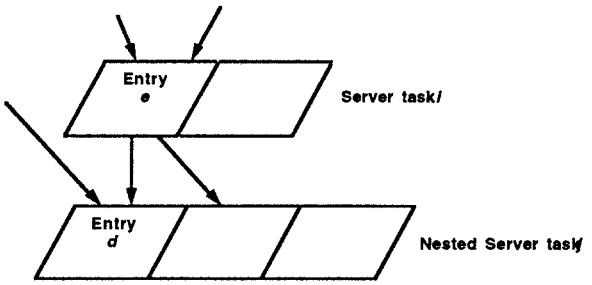


Figure 8. CS "Entry" Notation for Separate Services

The mean residual time of a phase depends on

- the mean and variance of the execution "slices" by the server task itself, in this phase
- the mean and variance of the gaps (nested services)
- the number of each of these (slices = gaps + 1).

The computation of V depends on the internal structure of a phase. Examples of phase structure for which V has been computed are:

- "stochastic" phase, with random number of each nested service (geometric distribution)
- "deterministic" phase, with a fixed number of each kind of nested service.

Standard techniques give the calculation.

Tools

Solution tools are available for these models. A heuristic MVA solver combines these equations in an iterative process. Petriu has improved the probability estimates by a detached Markovian analysis called "Task-Directed Aggregation" (TDA) (Aug. 91). Rolia has adapted standard queueing analysis techniques to solve one layer at a time, for systems with a strict layering [4].

The assumptions made by the analytic solvers are

- exponential "slice" execution times
- random or deterministic choice within a phase.

They are also capable of dealing with arrivals from outside at a stated rate, and with multiple copies of a task representing a set of identical clients, making up a server pool. In the "deterministic choice" model the service times can have general distributions.

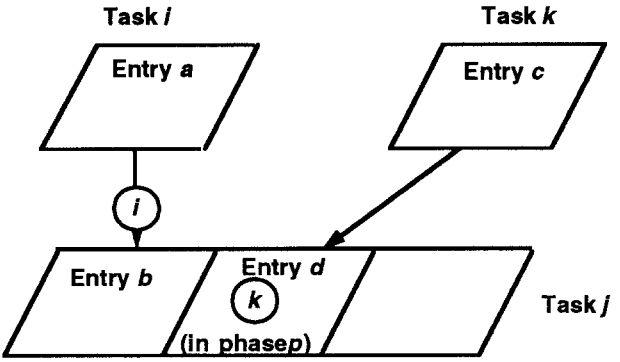


Figure 9. (a) An illustration of the arrival state InService(a,b,c,d,p)

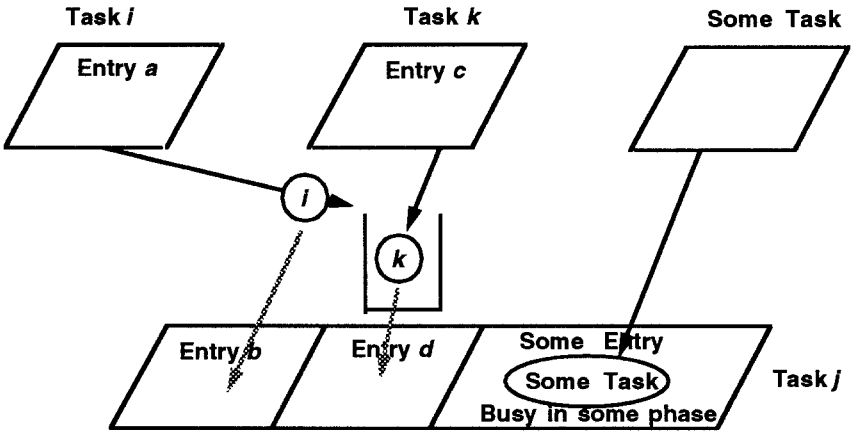


Figure 9. (b) An illustration of the arrival state InQueue(a,b,c,d)

Client i makes requests to Entry i of the server. Client i has one phase of length $s_{1,i}$ and entry i of the server has two phases of length $s_{2,i}$ and $s_{2,i}$.

Figure 10(a). Basic Clients and One Server

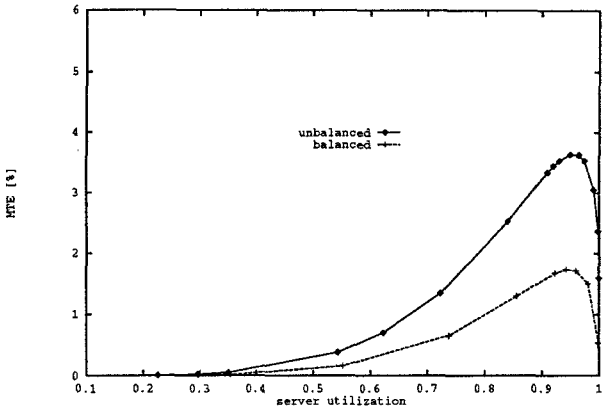


Figure 10(b). Errors for Various Server Utilizations

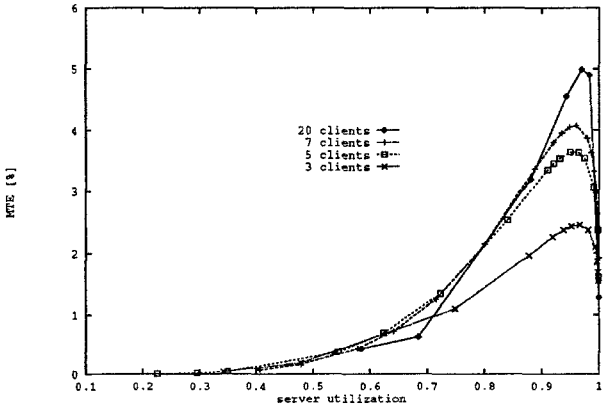


Figure 10(c). Errors for Various Numbers of Clients

Figure 10. Clients with separate service parameters (Entries)

Example: Basic Clients and One Server

A basic system with N clients and one server is shown in Figure 10(a), a separate entry for each client. The error of analytic modelling depends on the server utilization as shown in Figure 10(b). The Figure shows the magnitude of the largest throughput error among the N clients - the average error was much smaller. The results were found using TDA, which is exact for 2 clients. Figure 10(b) is for 5 clients, and for "balanced" service (all entries equal) and "unbalanced" (with service parameters varying over a range of an order of magnitude). Figure 10(c) is for various numbers of "unbalanced" clients, up to 20.

The outstanding features of these results are

- error increases with increasing server utilization, and then falls to a small value, with a peak value at around 95% - 98% utilization;
- error increases with the number of clients;
- error is 5% or less - mostly much less.

Example: "Directory Service"

Figure 11 shows a small information system that was implemented in a multiprocessor testbed running to local servers either to retrieve data by a GET entry, or to enter data by UPDATE. It is a tiny version of a set of directory servers, some with local information, and some with information for a wider area. Data was stored in in-memory tables, and measured execution times are given in "ticks" equal to 50.5 μsec. Using the measured service parameters, the model predicted client throughputs as shown, when the client "think time" x was varied. The client execution time was controlled by using a dummy loop. The error against the measured values again increased with load, since the load on the server is greater when x is smaller.

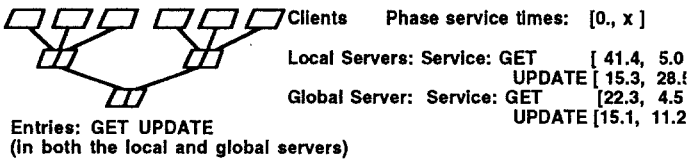


Figure 11. A Tiny Example Structured like a Directory Hierarchy

Notice that the measured service times did not have exponential distributions, shown by the coefficients of variation being much less than unity (ranging down to 0.04). This accounts for some of the error.

5.0 The Nature of Software Bottlenecks

Section 3 described how server saturation "pushes up" on higher-level processes, producing bottlenecks above them. A software server can be saturated even when its processor is not saturated, by dividing its service periods among requests for a variety of lower servers.

This section studies this effect through an example, beginning with a database server shown in Figure 14. It takes 20 time units to serve a request. Each of its 10 clients takes only 10 units to prepare the next request, so the database server is saturated. As it has its own processor that too is saturated (at least as the model is shown in Figure 14) it is a main-memory database with no secondary storage.

To relieve the bottleneck, the parts requiring mutual exclusion were factored out and re-allocated to three separate server tasks, each with a processor of its own, showin in Figure 15. The request goes to a dispatcher task which routes the parts to the servers, in order. There is shown no additional overhead for controlling the more complex execution path, and messaging; there is still 20 units of work per request, 2 units at the dispatcher and 6 at each server, in various combinations of first and second phases. Now the dispatcher can be run in multiple copiesl or clones, allowing requests to run in parallel.

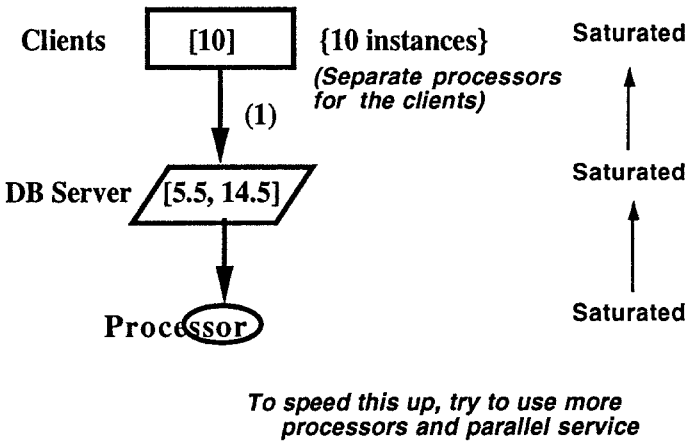


Figure 12. Original Case: Saturated DB Server

Figure 16 shows the effect of varying the number of clones. Perfect speedup with 4 processors and 20 sec of work per response would be 4x0.05, or 0.2 responses/sec; 50 clones give 0.156/sec. As there are 10 clients, it might be supposed that 10 clones would give all the available benefit, but the various second phases in the software make further gains possible up to 50 clones.

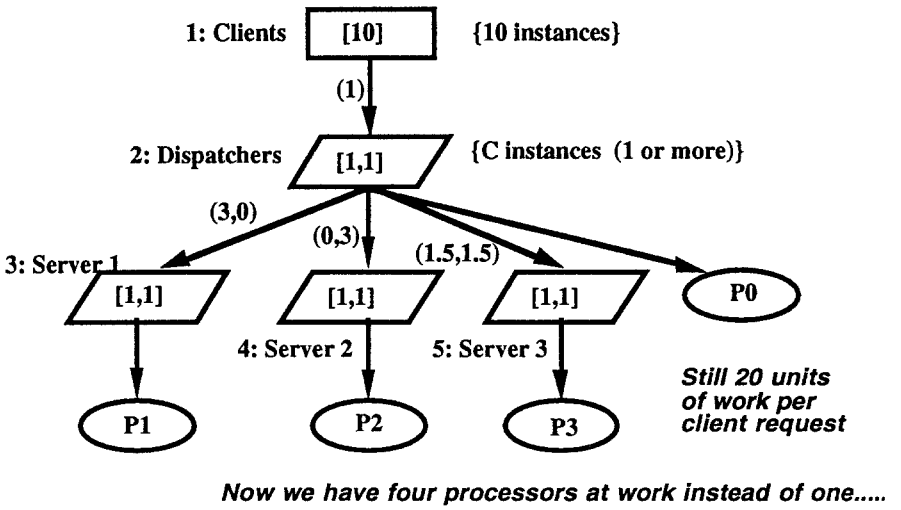


Figure 13. Dividing the Bottleneck Work, with three Servers for Critical Sections Parameters for the Base Case

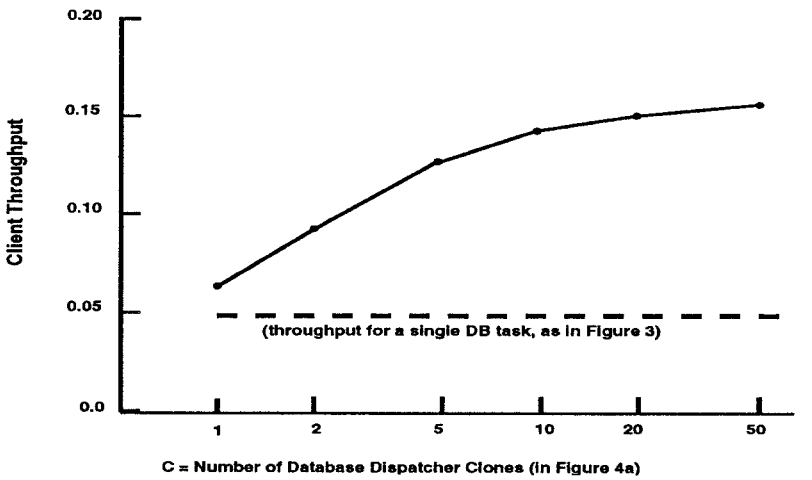


Figure 4c Base Case Client Throughput against Cloning Level

Figure 14. Results of Cloning the Dispatcher in the Base Case

Table 1. Full Results for the Base Case

Cloning Level	1	2	5	10	20	50
Client Throughput	0.065	0.093	0.126	0.143	0.152	0.156
Dispatcher Cycle	15.2	21.4	39.6	69.9	109.4	153.4
Dispatcher Queueing	135	86.6	49.4	25.8	7.9	0.74
Server 1 Queueing	0.54	1.23	3.27	6.38	8.35	8.85
Server 2 Queueing	0.54	1.22	3.25	6.73	13.0	24.3
Server 3 Queueing	0.36	1.00	2.93	6.13	11.3	14.2
Dispatcher Utilization:						
Task (each clone)	1.0	0.999	0.999	0.990	0.836	0.478
Processor	0.128	0.182	0.247	0.280	0.298	0.307
Server 1 Utilization	0.393	0.560	0.755	0.854	0.915	0.937
Server 2 Utilization	0.393	0.556	0.755	0.862	0.912	0.935
Server 3 Utilization	0.394	0.558	0.756	0.860	0.916	0.936
Bottleneck Strength	2.54	1.78	1.32	1.15	0.91	0.51
Actual ratio	2.40	1.68	1.23	1.09	1.03	1.03

Table 1 shows detailed results, including the task queueing delays and server utilization, found by simulation.

Software Bottleneck Strengh

We will now provide an operational definition of a software bottleneck:

A software bottleneck occurs when a task or its clone set exhibits a high utilization which is also high relative to the utilizations of all of its servers, either direct or indirect.

The second clause is necessary to differentiate between those tasks which may show high utilizations because of lower level bottlenecks and those which are at the root of the problem. It will be shown later that significant performance gains are possible only if the true bottleneck is identified and cloned.

In the Base Case with C = 1 the bottleneck is at the dispatcher because all three bottom level servers and its processor have lower utilizations. The bottleneck stays at the dispatcher up to C = 10.

The "bottleneck strength" of a task or its clone set is the ratio of its utilization to that of its most highly utilized server, direct or indirect.

If the bottleneck is somehow removed, and the limit is then determined by that most highly utilized server, then the bottleneck strength is also the ratio of the later throughput to the earlier one. To this extent the strength predicts the gain in performance through cloning. Let task b be the candidate bottleneck and U_i be the utilization of task i, for each task i which is a direct or indirect server of task b (including its processor). Then the bottleneck strength at task b is B_b defined by:

$$B_b = U_b / (\max_i U_i)$$

and the throughput after cloning task b cannot be greater than B_b times the value before cloning.

In the Base Case with C = 1, B = 2..54 and thru = 0.65; this predicts a potential throughput of 1.65, while the actual potential is 1.56. This is quite a close prediction.

Now returning to Table 1 we can calculate the bottleneck strength of the dispatches in the first column as BA=1.0/0.394=2.54. This is intended to predict the available gain due to cloning the administrator; if we examine the gain at 50 clones we find the throughput increase is a ratio of 0.156/0.065=2.40. More clones would perhaps give a little more gain, closer to 2.54; in any case it is close.

If we begin at 10 clones the measured bottleneck strength is only 1.15 and the experimentally obtained gain is 1.09 - again close.

Just in case this close agreement was due to the particular pattern of first and second phases the entire experiment was repeated for several other cases including all phase-1 - with similar results.

Table 2. The Nature of Software Bottlenecks

For the familiar bottleneck in a queueing network:	**For a "Software Bottleneck":**
• typically one saturated resource	• several, due to pushback
• relief by	
- speeding it up	- less effective
- adding resources	- YES
• throughput increase is locally proportional to	
- speedup at the sat, point	- less than proportional
- added resources	- YES, proportional
• bottleneck location is determined by a simple analysis of the parameters	• NO, requires a contention analysis

Table 2 summarizes the differences we see between the familiar "hardware" bottleneck, such as a processor or i/o device, and a software bottleneck as defined here. The reason that code speeding gives less than proportional system speedup is that the saturated server

spends only part of the time executing its own code; the rest is spent blocked. Speedups which result in fewer requests will also be helpful but are not in the Table.

6.0 Conclusions

In this tutorial

- Service times in software servers were characterized in terms of nested services.
- "Rendezvous Networks" notation provides a compact view of the workload and communications relationships of client-server systems.
- Solution techniques were described based on queueing analysis.
- Locating *Software Bottlenecks* often requires a contention analysis.

7.0 Bibliography

[1] W. Rosenberry and D. Kenney, "Understanding DCE", published by O'Reilly & Associates, Inc.,103 Morris Street, Suite A, Sebastopol, CA 95472, U.S.A.

[2] C. M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", *Performance Evaluation*, vol. 9, n. 2, pp. 143-160, April 1989.

[3] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Multi-Tasking Distributed Software, *IEEE Transactions on Computer*, under review.

[4] J. A. Rolia, "Predicting the Performance of Software Systems", PhD thesis, University of Toronto, January 1992.