Software Bottlenecking in Client-Server Systems and Rendezvous Networks

J.E. Neilson   C.M. Woodside   D.C. Petriu   S. Majumdar
Real-Time and Distributed Systems Group
Carleton University, Ottawa K1S 5B6
e-mail: neilson@scs.carleton.ca, or cmw@sce.carleton.ca
Jan 26, 1994

Abstract

Software Bottlenecks are performance constraints caused by slow execution of a software task. In typical client-server systems a client task must wait in a blocked state for the server task to respond to its requests, so a saturated server will slow down all its clients. A Rendezvous Network generalizes this relationship to multiple layers of servers with send-and-wait interactions (rendezvous), a two-phase model of task behaviour, and to a unified model for hardware and software contention. Software bottlenecks have different symptoms, different behaviour when the system is altered, and a different cure from the conventional bottlenecks seen in queueing network models of computer systems, caused by hardware limits. The differences are due to the "push-back" effect of the rendezvous, which spreads the saturation of a server to its clients. The paper describes software bottlenecks by examples, gives a definition, shows how they can be located and alleviated, and gives a method for estimating the performance benefit to be obtained. Ultimately, if all the software bottlenecks can be removed, the performance limit will be due to a conventional hardware bottleneck.

1. Introduction

Client-server architectures are widely used in transaction processing and in modern operating systems, and are implicit in proposals for open distributed processing such as DCE. In those systems where, after making its request, the sender must wait for the reply in a send-blocked state, we will term the request-wait-reply pattern a *rendezvous.* A special model, the Rendezvous Network or Layered Queueing model [Wood86, MWNP88, WNPM89,Rol91] was formulated for these systems with blocking including client-server interactions as shown in Figure 1a below, and multiple levels as in Figure 1b. Many important classes of systems conform to the Rendezvous Network model, including systems using Remote Procedure Calls (RPC) [RPC], open distributed processing systems, the "Non-stop" system sold by Tandem Computer [Bart81], systems using concurrent object-oriented frameworks such as Actors [Agha86], those based on real-time kernels like QNX or Amoeba which use a blocking send for inter-process communications [Quan91, Tanb90] and those programmed in Occam or Concurrent C for the Transputer [INMOS88], and systems using Ada tasking. Although we are mainly interested in application-level servers, exactly the same concerns arise in operating systems based on servers. Among the advantages of the rendezvous is efficiency, since it provides communication without the effort of buffer management and the message copying associated with asynchronous communication. However some potential for concurrency is lost, and there may be performance-impairing bottlenecks when a key task spends long periods send-blocked. These bottlenecks are the subject of the present paper.

A bottleneck is a single contention point that limits the total system performance. Device bottlenecks, at a disk, a cpu or a bus, are familiar in performance analysis (e.g. [LZGS84]) and are easy to identify, because the bottleneck becomes saturated, while other devices do not. Other kinds of bottlenecks may be due to execution memory or buffer space, critical sections, locking of data, blocking of processes, and overloaded server tasks, and give more complex behaviour due to simultaneous resource possession. The present study examines process blocking and server overload in the particular case of processes with client-server relationships. An overloaded software server tends to "push back" on its clients, which in turn appear to be loaded as well. In a deeply layered system, software blocking spreads upwards and outwards and may affect a large part of the system, while at the same time the hardware resources may all be underutilized. This phenomenon seems to be a familiar one in databases, where it is counteracted (as here) by providing parallel "clone" processes to spread the load at the bottleneck. We have not however discovered much formal analysis of this problem. Up to this time there has been only a very limited case considered in [Font89], and a simple performance bound calculation for Rendezvous Networks given in [Wood86].

The growing importance of client-server systems in practice motivates the present effort to obtain some general results for diagnosing software bottlenecks, guidelines for relieving them by multi-threading or "cloning", and estimates of the potential improvement. It is well known, especially in the database and operating system fields that sufficient processes need to be provided for various services, to avoid limiting throughput. Cloning is a standard way to deal with server constraints, because as a purely software solution it is cheap. In UNIX RPC servers it is common to fork a task to serve a request. If this is done without bound however it may cause thrashing, or it may even fail on a memory limit. In DCE it is possible to fork a new thread for every request, or to run a fixed number of static threads and queue additional requests when necessary; the latter option avoids thread-creation overhead. In Tandem Non-Stop (TM) a fixed set of static clones is used but the number can (optionally) be varied dynamically. This paper assumes a finite set of statically created clones. It applies also to dynamically forking server clones, when there is a limit on the number of clones.

The approach of the present paper is informal, obtaining insight through simulations of several client-server systems. The results are intuitively reasonable. For simplicity we ignore the communications delay between processors in the body of the paper. However, this and some other limitations in the model can easily be overcome, as indicated in the last section. In the next section the problem is stated in the general framework of the Rendezvous Network model, which accommodates a client-server architecture and both deterministic and stochastic task behaviour. Section 3 investigates software bottlenecks, leading to some general observations, a definition of a software bottleneck and a measure of its intensity ("bottleneck strength"). It shows how bottlenecks arise, how they can be relieved by cloning, and how the improvement is limited by other potential bottlenecks lying behind the first. Ultimately the software bottlenecks can be removed, leaving only hardware-based limitations. Section 4 extends the discussion to a variety of other intertask communications patterns, and to more deterministic patterns of behaviour. Section 5 gives a simplified but less accurate analysis, and Section 6 gives conclusions and describes some generalizations of the model.

2. The Rendezvous Network Model

A Rendezvous Network describes "tasks" which may represent software processes (clients and servers), the users of the system, and the hardware devices. Tasks demand service from each other. Figure 1a shows the simplest model with clients and one server, while Figure 1b shows a multi-layer model in which servers like Task 3 make requests to other lower-level servers such as Task 4, and also to a processor "tasks". Where a processor which has only a single software task they are usually merged together to simplify the model, as is the case for Task 1. The graph is acyclic, to avoid waiting cycles which might cause deadlock.

We are most interested in the throughput capacity and delay to the client tasks at the top of the graph, called "reference" tasks, which are assumed to loop forever, taking time and generating requests. Figure 2 illustrates a trace of behaviour for Figure 1b. A task which serves requests (such as Tasks 3, 4, 5) may have a first phase of service before sending the reply, and an optional second phase after the reply. While in the first phase (in the rendezvous) the client is blocked and the server merely continues the thread of control of its client, but in the second phase it has an independent thread of control of its own. A reference task has only a sequence of second phases, as it accepts no requests.

Formal Assumptions and Parameters

The parameters labelled on the graph of tasks and their interactions (such as Figure 1b) are

- T_i, on reference tasks (the rectangles). T_i is the mean delay time per cycle.
- s_{ip}, on server task i, in the form $[s_{i1},s_{i2}]$. s_{ip} is the mean total execution time in phase p, for p=1,2.
- y_{ijp} on an arc from task i to j, in the form (y_{ij2}) if i is a reference task or (y_{ij1}, y_{ij2}) if i is a server task. y_{ijp} is the mean number of requests by rendezvous from task i to task j, during phase p. For a reference task there is only y_{ij2} , since it only has a second phase.

Referring to Figure 1b, these definitions are interpreted to make $T_1 = 4$, $s_{12} = 3$, $y_{132} = 0.3$, $s_{31} = 0.1$, $s_{32} = 0.7$, $y_{341} = 0.8$, etc. The processor service times and request rates are not shown; they are derived from the execution time of the tasks, and the number of requests they make, in the obvious way as described in [MWNP88]. In the mathematical analysis of these networks it is assumed that the intervals of execution time between requests are random, independent and exponentially distributed, and that the requests per phase are random and geometrically distributed. This is meant as an abstraction for data-dependent execution and request sequences.

In the present work simulation has been used to obtain performance values because the available approximations (given in [Wood86], [MWNP88], [WNPM89], [Petr91], and [Rol91] for example) do not cover the use of clone-sets for multi-threaded tasks. The simulation results were found with 95% confidence intervals of better than +/– 1% of the stated values, by using a number (at least 10) of replicated runs, with 10% of the run-length used as start-up time to reach steady state.

Clone-Sets

This work investigates bottleneck relief using "clone-sets" consisting of multiple identical copies (i.e. clones) of some tasks. The tasks in the set receive requests from the same set of clients and send requests to the same set of servers; they also run on the same processor, or on a processor clone-set which could represent a multiprocessor. A clone-set at task i is a multiserver, with C_i identical servers which are all copies of a class definition for task i, and a common FIFO queue. A reference-task clone-set represents C_i identical reference tasks.

3. The Software Bottleneck Phenomenon

The emergence of a software bottleneck will now be described by an example in Figures 3 and 4. Figure 3 shows 10 client tasks with a cycle including 10 units of delay and one request to the server, which in turn has a 20-unit execution time (5.5 units in phase 1, 14.5 in phase 2). The server runs on a processor shown as an ellipse. Naturally the server is saturated, and runs at its maximum rate of 0.05 responses/unit time. This is device saturation. To speed up the system the server is divided into four parts in the "Base Case" in Figure 4a, running on four processors. Three lower level single-threaded servers provide three parts requiring mutual exclusion, and a multi threaded "dispatcher" task provides overall control. As shown, the parameters are:

$$T_1 = 10;\ y_{122} = 1;\ s_{ip} = 1 \text{ for all } i = 2.5 \text{ and } p = 1,2;\ y_{231} = y_{242} = 3;\ y_{251} = y_{252} = 1.5.$$

The average load for each client response cycle is 2 units of execution at the dispatcher and 6 units at each of servers 1, 2 and 3. A consideration of device bottlenecks would suggest that the client throughput will now be limited at 1/6 = 0.167 by saturation of the new servers, and that the dispatcher will be lightly loaded. Because of blocking however, the dispatcher is still the bottleneck and the throughput rises very little, only to 0.065. In discussing task saturation we will define the *utilization of a task* to be the fraction of time when it is busy or blocked in a rendezvous. We may consider that the dispatcher task divides its service cycle among four servers, spending 2 units at its processor (its execution time) and the remainder at the three software servers, blocked. This example motivates the following general observation:

Observation 1: When a task is a software bottleneck it is saturated but its servers are not.

Next consider the changes that occur if the execution time of server 1 increases. The system slows down and the bottleneck shifts gradually down to server 1 and its processor. Figure 4b shows this shift, starting from the Base Case at point *X*. At point *Y* the shift is virtually complete and P1 (as well as server 1) is 89% utilized; there is now a processor bottleneck at P1. The dispatcher is still 100% saturated at point *Y* but it is no longer a software bottleneck because it has a saturated server, a candidate for a faster processor P1. The gradual change in load just described also shows how saturation spreads in tasking systems by the "push-back" effect of rendezvous communication.

Observation 2: A saturated task or processor tends to saturate tasks which use it by rendezvous, so bottlenecks tend to spread saturation in a rendezvous net back along the request arcs.

Naturally, saturation spreads back more readily to tasks which are heavy requesters, than to those which are not. If queues become long enough however, all requesters are eventually affected.

Cloning

Within the system of Figure 4a, the cloning level C of the dispatcher task was increased in steps, giving the throughputs shown in Table 1 and Figure 4c. The Figure shows that at C=10 most of the potential gain has been achieved, the original throughput of 0.05 has been nearly tripled, and the limit to further increase is the server tasks and processors, which are at nearly 90% utilization. The bottleneck has been pushed down to the processors P1 - P3. The dispatcher task utilization is still high however, and we view this as *transmitting* the saturation of P1 - P3 upwards to the clients. Increasing *C* to 50 gives little further change.

Observation 3: A sufficient level of cloning at a bottleneck may push the bottleneck downwards to its servers, by increasing their load. In such a case the clone-set still operates to propagate saturation upwards to its own users.

In other situations the increased throughputs may cause the bottleneck to migrate upwards or even sideways. We will now provide an operational definition of a software bottleneck:

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

The second clause in the definition 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. The bottleneck strength is only a rough indicator, and significant performance gains are possible only if the true bottleneck is identified and cloned. A software bottleneck condition arises most readily in cases where there is a balanced 'fan-out' of service requests over a number of servers. However it can arise with just one software server. Since the processor of the bottleneck task is also a kind of server, the bottleneck strength of a task with just one software server could be as high as 2.

Estimating the Potential Gain from Bottleneck Strength

The maximum performance gain achievable, if a software bottleneck is eliminated through cloning, is limited either by a new bottleneck (if the load is sufficient), or by the offered load (the reference tasks). Given sufficient load and provided all software bottlenecking can be eliminated, throughput performance will be governed by whichever processor (or leaf task) saturates first. If the total mean service requirements per client transaction are known for each processor, the processor with the greatest total service determines the maximum achievable throughput. In the Base Case the throughput limit is therefore 1/6, which is approached closely in Figure 4c.

A metric for a bottleneck can be derived from the operational definition. *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 a fully saturated bottleneck is relieved by cloning and the new bottleneck is that most-highly-utilized server, then the throughput may increase by a factor up to the bottleneck strength if the service time is load-independent. Note however that (1) if the service time increases with throughput the gain will be smaller, and (2) the new bottleneck may be elsewhere. For example, the Base Case value for the dispatcher task was 2.54 which predicts an improved throughput bound of $2.54 \times 0.05 = 0.165$, very close to the value achieved with 50 clones.

Formally, let task b be the candidate bottleneck, and U_i be the utilization of task i, for each task 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. It may be limited at a lower value if the task utilizations are load-sensitive (as in the example with application-level tasks, below).

4. Variations on the Base Case

The rules of thumb developed in Section 3 will now be explored in some variations on the Base Case, with different service patterns and more elaborate task structures. This will show that the results are not due to the particular parameters in the example used so far. The variations were called RPC, Update, Non-Homogeneous and More Centralized, with the parameter values given in Table 2. The servers S1 - S3 in the “RPC Case” have only phase-1 service (like the most basic kind of remote procedure call). In the “Update Case” the servers are phase 2 dominant, which gives higher potential concurrency. The Non-Homogeneous Case has three distinctly different servers; one with dominant phase 1, one with dominant phase 2, and one balanced. The More Centralized Case in the last column is described separately, later.

Figure 5 shows the throughput results. The first three cases have the same bound of 1/6, due to the servers, which they all approach quite closely as dispatcher cloning levels increase. We present detailed results for only the RPC Case (see Table 3) since phase-1 servers are particularly common. Notice that the throughput/cloning relationship is slightly more dramatic here than for the base case, and the bottleneck strength is also greater. Overall, however, it behaves very much like the Base Case, particularly for high levels of dispatcher cloning.

Similar throughput results may be seen in Figure 5 for the Update and Non Homogeneous Cases. The latter was particularly close to the Base Case with client throughputs increasing from 0.064 with a single dispatcher up to 0.153 with 50. In the Update Case the bottleneck strength was lower (at 2.09) with a single dispatcher, and the throughput did not approach the bound as closely as the others. This suggests that a more inherently concurrent model, such as the Update Case with dominant second phase server service, is both less inclined to bottleneck and less inclined to respond to the cloning treatment. Table 4 shows the value of B at $C = 1$ and the actual improvement at $C = 50$, for all the cases.

The “More Centralized” Case is different from the rest, in having a dominant first phase. Eight of the 20 units of execution per response is at the dispatcher, and 4 at each server. Table 4 shows that the processor saturation moves to P0, at a throughput level of 1/8, and is reached with only 10 clones. In this case cloning shifts the bottleneck from the task to its own supporting processor rather than to another task or processor.

Another Layer: Application Tasks

The next example shows how new bottlenecks can develop above a bottleneck when it is cloned, rather than below it. Figure 6 shows tasks between the clients and the dispatcher, representing information system application tasks that access the data base implemented by the dispatcher and the three servers. Every task runs on its own processor. This corresponds approximately to an actual database system we have studied, implemented in the Tandem "Non-stop SQL" environment [BoPu88, Zell90]. The Dispatcher layer corresponds to the SQL servers, and the Server layer corresponds with the disk access control servers. Simulation results with a single application and a single dispatcher indicate that the utilization of the application task is unity while that of the dispatcher is 0.959, and all the processor utilizations are less than 0.4.

It is tempting to call the application a software bottleneck (it has the highest utilization and the biggest waiting queue). However, cloning it gives only a tiny improvement in client throughput, up to 0.065 from 0.063. If the dispatcher task is cloned instead, client throughput rises more, to 0.081. The bottleneck strengths (1.04 for the application and 2.53 for the dispatcher) clearly and correctly point to the dispatcher as the bottleneck, while simple utilization data is misleading. However, to achieve the full indicated gain, both must be cloned. Table 6 indicates the effect of various cloning levels. As we move to the right in the table, adding application clones, the throughput rises and then saturates, indicating a dispatcher bottleneck. As we move down, adding dispatchers, it rises again and saturates, indicating that the bottleneck has moved back to the applications. Although the task utilization and bottleneck strength values are not shown in the table they consistently and correctly indicate the location of the bottleneck and which of the classes to clone next, for the greatest gain. That is, wherever the throughput increases more by going down (rather than to the right) in the table, the bottleneck strength of the dispatcher is greater, and vice versa. Finally, at 10 and 50 clones, the improvement factor is close to the predicted value of 2.53 (0.155/0.063 = 2.46).

A Deterministic Model

The above models all have stochastic execution times and service request patterns. To demonstrate that more deterministic systems have similar behaviour, cases were examined with low-variance execution times, including constant execution times, and with a fixed pattern of requests to servers 1 to 3. The results were very similar to those already seen.

5. A Quick Approximation

The simulation results used above required a significant effort to calculate. A quick approximation to estimate task utilizations will now be given, based on putting all a server's time into phase 1, and ignoring queueing delay. There must be only a single set of symmetrical reference tasks, all numbered 2 but treated as distinct. For every task, compute

- $S_i = s_{i,1} + s_{i,2}$ (or $S_i = T_i$ for $i = 1$)
- $Y_{i,j} = y_{i,j,1} + y_{i,j,2}$
- $X_i = S_i + \Sigma_j Y_{i,j} X_j$ = total "service" of task i

We require for the quick approximation there be just one group of reference tasks, numbered 1 without loss of generality. For each server task j, let its request frequency relative to one reference task response be a_j, found by $a_j = \Sigma_i\ Y_{ij}\ a_i$ and $a_1 = 1$. The throughput f_j for task j satisfies $a_j = f_j / f_1$. Then the load per client response on task i is $a_i X_i$, and on processor k it is $X_k = \Sigma a_i S_i$ summed over the tasks running on the processor. The processors will be numbered as additional "tasks" with artificial values $a_k = 1$. Then the saturation throughput f_1* for the reference tasks is determined by the largest value of $a_i X_i$, which gives f_1* for a situation with task i saturated:

$$f_1^* = 1/(\max_i a_i X_i) \text{ responses/sec}$$

and the task throughputs and utilizations are estimated as $f_1 = a_i f_1^*$, $U_1 = f_i X_i = f_1^* a_i X_i$. This is the quick approximation. For the Base Case example it gives:

- $X_3 = X_4 = X_5 = 2$; $X_2 = 20$, $X_1 = 30$.
- $a_1 = 1$, $a_2 = 10$, $a_3 = a_4 = a_5 = 30$,
- processors *P0, P1, P2, P3* are numbered 6, 7, 8, 9 respectively with $X_6 = 20$, $X_7 = X_8 = X_8 = X_9 = 60$,
- $\max_i a_i X_\text{i} = 200$ for the dispatcher; $f_1^* = 1/200$ (per client), or 1/20 total
- $U_1 = 3/20$, $U_2 = 1$, $U_3 = U_4 = U_5 = 6/20$
- Bottleneck strength at the dispatcher = 20/6 = 3.33

Although the phase-1 assumption, and ignoring queueing, gives an erroneous estimate of f_1* it does locate the bottleneck correctly at the dispatcher. The strength estimate 3.33 is close enough to the simulation value 2.54 for many practical purposes.

The other cases in Section 5 give the same value $B = 3.33$ except for the "More Centralized" Case. It has a larger dispatcher service time $S_2 = 8$ and smaller server times of $S_3 = S_4 = S_5 = 4$, with $a_2 = a_3 = a_4 = a_5 = 10$. Briefly, the values of $a_i X_i$, including the processors, are (30, 200, 40, 40, 40, 80, 40, 40, 40) so $f_1^* = 1/200$ and $B = 2.5$. The dispatcher processor is the new bottleneck.

Referring back to the results shown in Table 4 for the actual bottleneck strength and the throughput improvement, the quick approximation is excellent in the RPC Case where the phase-1 approximation is exact, poorer in the Base Case, and quite poor in the Update Case which is mostly phase 2. Clearly it should be used with caution.

6. Conclusions

The bottleneck definition developed in Section 3 identifies a bottleneck caused by software limitations. Its strength can then be calculated, and indicates the potential value of removing it. Note however that other paths in the system may limit the actual improvement obtainable. When system measurements or simulation data are not available one can use the simpler but more approximate calculation described in Section 5.

To remove bottlenecks we have considered multi-threaded execution in the form of "task clones", which are multiple instances of a single task class. In the right circumstances the number of clones of the bottleneck task which have a significant effect on throughput is approximately the same (within a factor of two) as its bottleneck strength. However the bottleneck tends to shift to another task, which may be below the original task in the server hierarchy (as in Section 3), above it (as in the first example of Section 4.2, see Figure 5), or "beside" it, if there are other access paths besides the one with the bottleneck. It may be necessary to clone several tasks to achieve the maximum performance, which ultimately is limited by the processors.

In practice the abstract system modelled here can appear in various guises. The limitation modelled here by single-threaded tasks with rendezvous may be either due to the style of programming used, or may be intrinsic to the operating environment. Similarly the solution by cloning may be implemented by creating multiple instances of a task class, or by using multiple threads within a single server process; the performance effects are much the same.

Software bottlenecks combine two phenomena which have been studied at length, namely *blocking* and *simultaneous resource possession.* However, the nature of the blocking is quite different. The usual queueing system with blocking (see e.g. [AKPE89]) blocks a server because of lack of buffer space, rather than intentionally (and always) waiting for another service to complete. And the customers usually proceed onwards in the net, rather than traversing a tree as they do here. Blocking in software and buffer blocking do have in common the effect of saturation "pushing back" to block other resources. Simultaneous resource possession, first described in [JALA82], is the essence of a software bottleneck. A client "holds" software task resources along the path of a request, as long as they are in phase 1. Because of "pushing back", several resources along the path may become saturated. The combination of blocking, pushing back and simultaneous possession which appears in software bottlenecks is quite distinctive.

The discussion of the paper has been simplified in a number of ways, and the definitions given here can be applied without change to a broader class of systems. The tasks may have separate entries or ports for different services, with different execution parameters, and queues may have priorities [WNPM89]. There may be more complex architectures with servers representing buses or network connections, so that communications bottlenecks may also be detected [WMNP91]. The reference tasks may include an execution phase allocated to a processor where it contends with other tasks, which may be at any level in the hierarchy. There may be groups of different kinds of reference tasks, corresponding to different classes of users in other models (Section 5 would not them apply). There may be "open" arrivals of transactions at a fixed rate, rather than transactions generated by the reference tasks. With these generalizations a wide range of systems can be analyzed.

Acknowledgments

The anonymous referees made many helpful suggestions. The support of the Telecom Research Institute of Ontario and of the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

References

[Agha86] G. Agha, *"Actors: A Model of Concurrent Computation in Distributed Systems",* MIT Press, Cambridge, Mass., 1986.

[AKPE89] I. F. Akyildia, H. G. Perros, editors, Special issue of *Performance Evaluation* on Finite Queues with Blocking, v. 10, n.3, 1989.

[Bart81] J. F. Bartlett, "A nonstop kernel", *Proc Eighth Symp. on Operating Systems Principles*, ACM, 1981.

[BoPu88] A. J. Borr, F. Putzolu, "High Performance SQL Through Low-Level Integration", in Proc. SIGMOD 88, *ACM SIGMOD Record*, v 17, n 3, pp 342-349, Sept. 1988.

[Font89] M. J. Fontenot, "Software congestion, mobile servers, and the hyperbolic model", *IEEE Trans. on Software Engineering*, v 15, pp 947-962, 1989.

[INMOS88] *"Occam 2 Reference Manual"*, INMOS Ltd., 66 Wood Land End, Hemel Hempstead, England HP2 4RG, Prentice-Hall, 1988.

[LZGS84] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, *"Quantitative System Performance"*, Prentice-Hall, 1984.

[JALA82] P. A. Jacobson, E. D. Lazowska, "Analyzing Queueing Networks with Simultaneous Resource Possession", *Commun. of the ACM*, vol 25, number 2, pp. 142-151, February 1982.

[MWNP88] J. W. Miernik, C. M. Woodside, J. E. Neilson, D. C. Petriu, "Throughput of stochastic rendezvous networks with caller-specific service and processor contention", *Proc. IEEE Infocom 88*, pp 1040-1049, Mar. 1988.

[Munt74] R. R. Muntz, J. W. Wong, "Asymptotic properties of closed queueing network models", *Proc 8th Princeton Conf. on Information Sciences and Systems*, 1974.

[Petr91] D. C. Petriu, C. M. Woodside, "Approximate MVA from markov model of software Client/server systems", *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, Dallas, Dec. 1991.

[Quan91] *"QNX 4.0 User's Guide"*, Quantum Software Systems, Kanata, Ontario, April 1991.

[Rol91] *"Predicting the Performance of Software Systems",* Ph.D. thesis, Univ. of Toronto, May 1991.

[Tanb90] A. S. Tanenbaum et al, "Experiences with the Amoeba Distributed Operating System", *CACM* v. 33, Dec 1990.

[Wood86] C. M. Woodside, "Throughput calculation of basic stochastic rendezvous networks", *Performance Evaluation*, v 9, pp 143-160, 1989.

[WNPM89] C. M. Woodside, J. E. Neilson, D. C. Petriu, S. Majumdar, *"The stochastic rendezvous network model for performance of synchronous multitasking distributed software",* Report SCE-89-8, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, revised August 1991.

[WMNP91] C. M. Woodside, S. Majumdar, J. E. Neilson, D. C. Petriu, *"A Guide to Rendezvous Network Modelling of Client-Server Software Systems"*, Report SCE-91-22, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, April 1991.

[Zell90] H. Zeller, "Parallel query execution in Nonstop SQL", *IEEE Spring COMPCON 90 Digest of Papers*, pp 484-487, Feb. 1990.

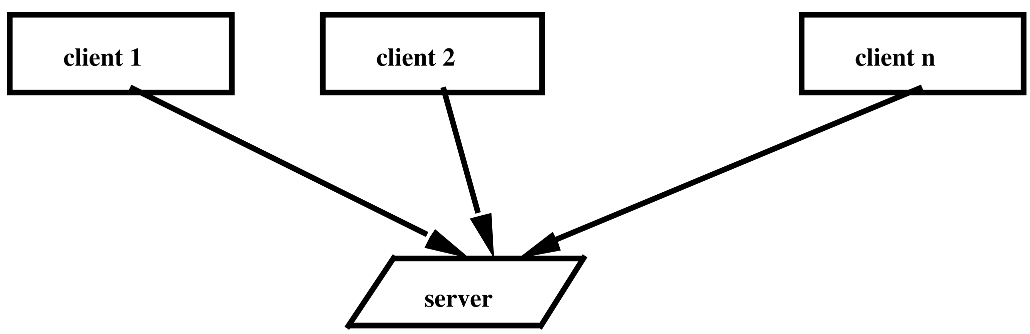

Figure 1a Simple Client Server Rendezvous Net

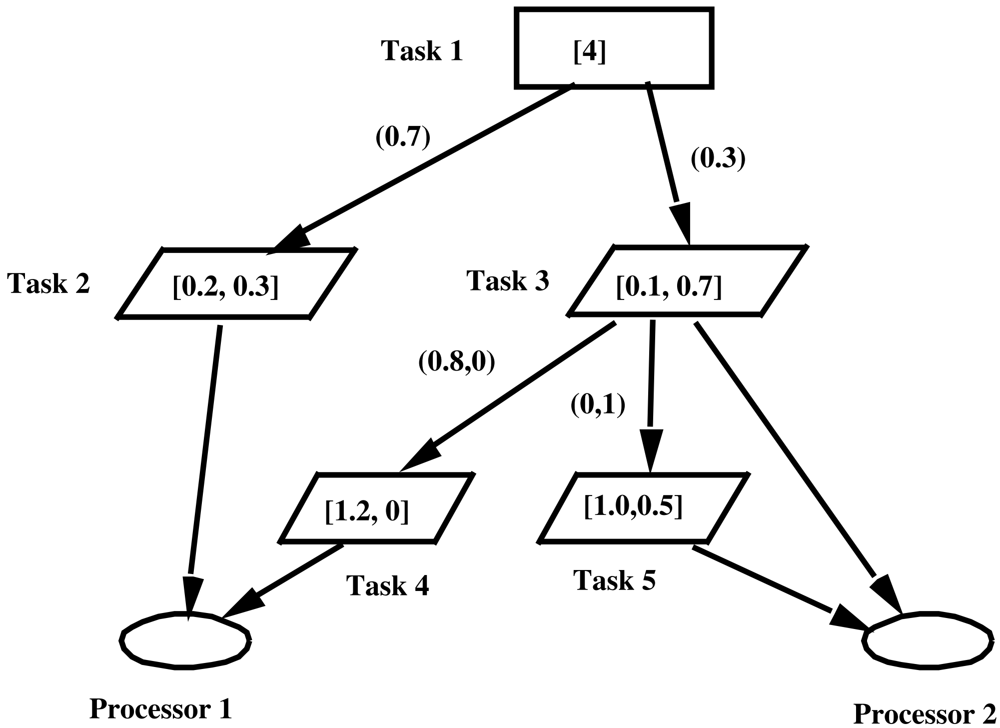

Figure 1b   Example Stochastic Rendezvous Network with Multiple Layers

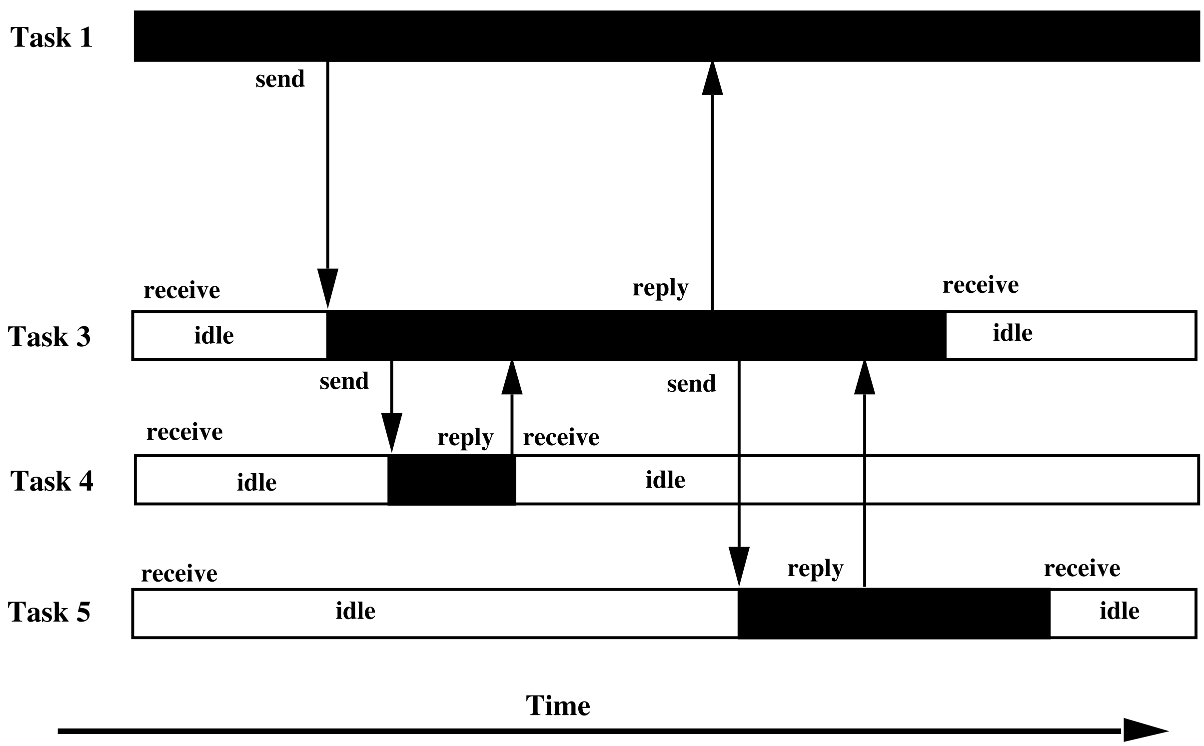

Figure 2 - Timing Diagram of Rendezvous Duty Cycles

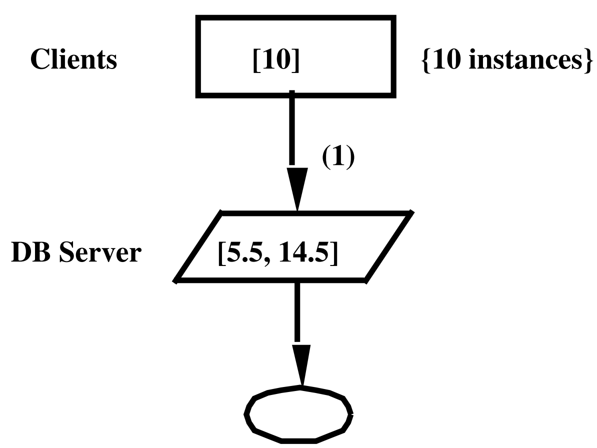

Figure 3 Database Client Server System

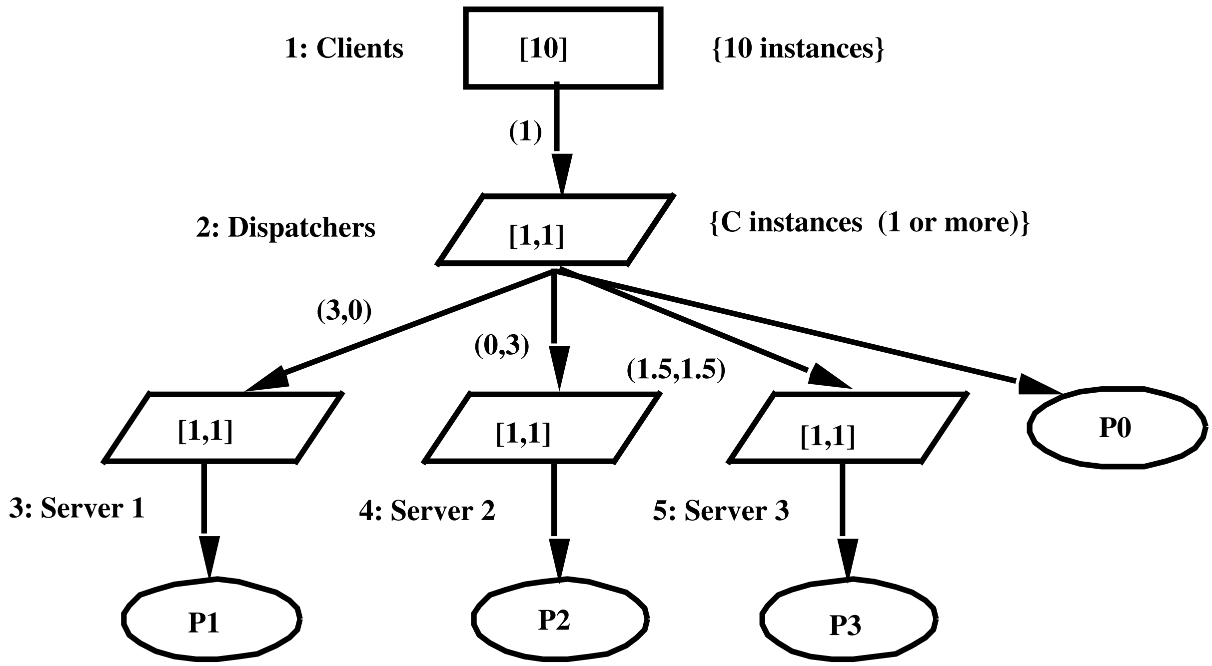

Figure 4a Distributed Database Server: the Base Case

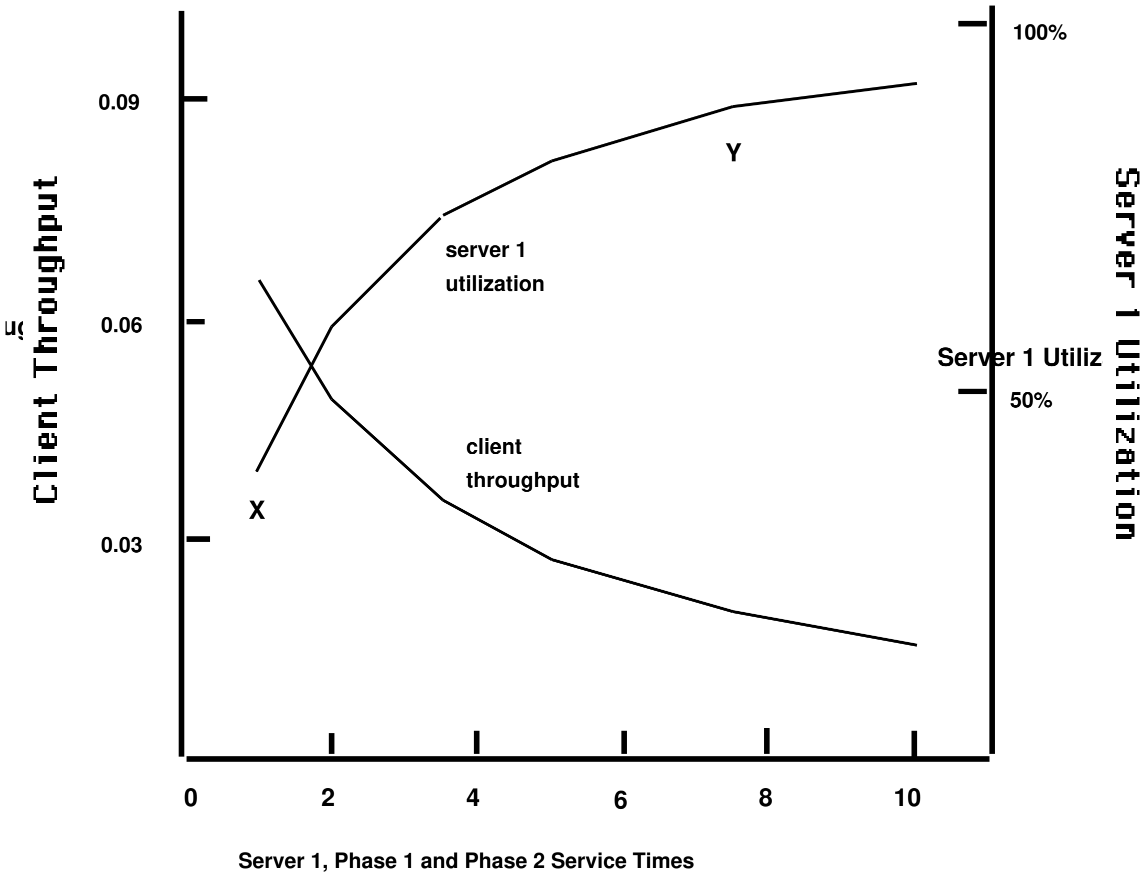

Figure 4b Moving the Bottleneck Down to Server 1

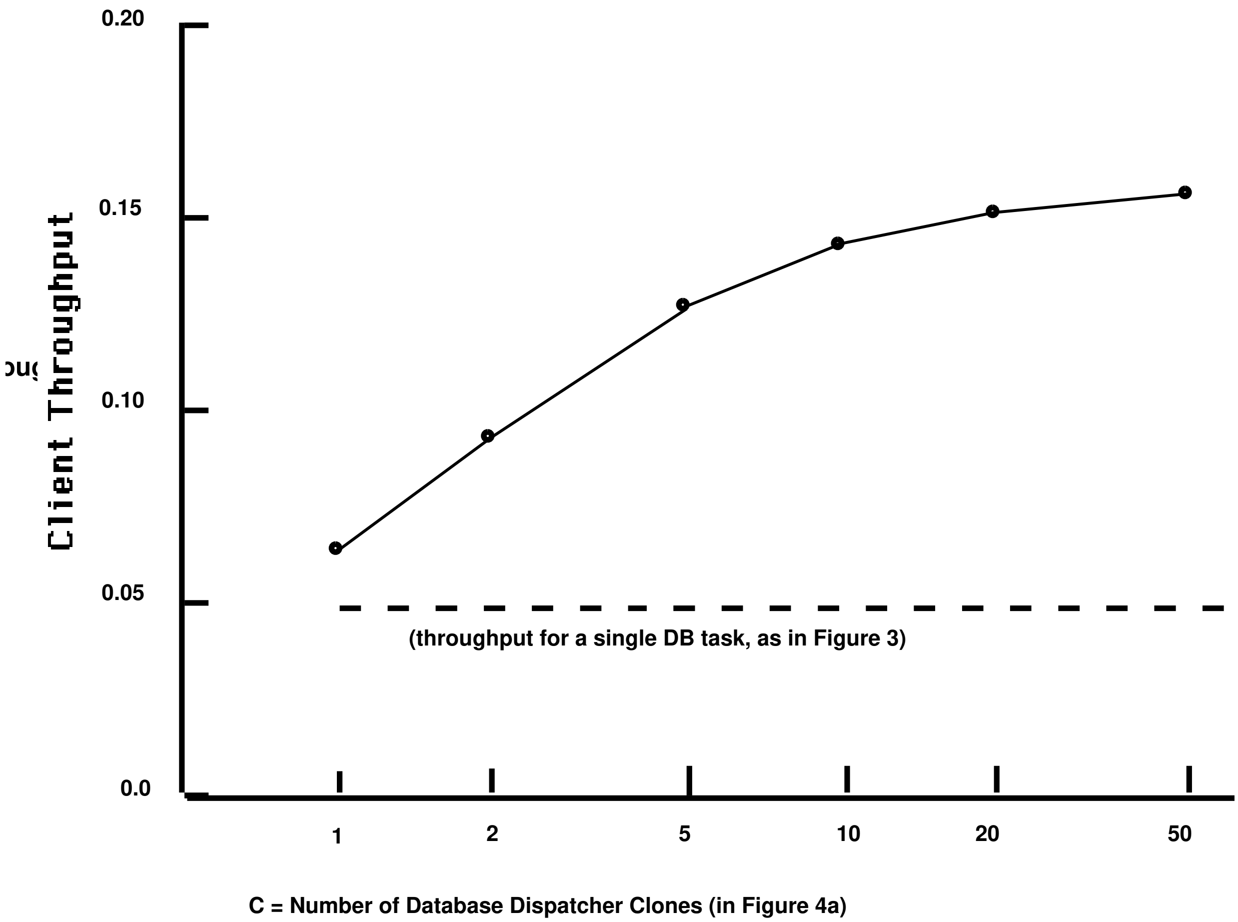

Figure 4c Base Case Client Throughput against Cloning Level

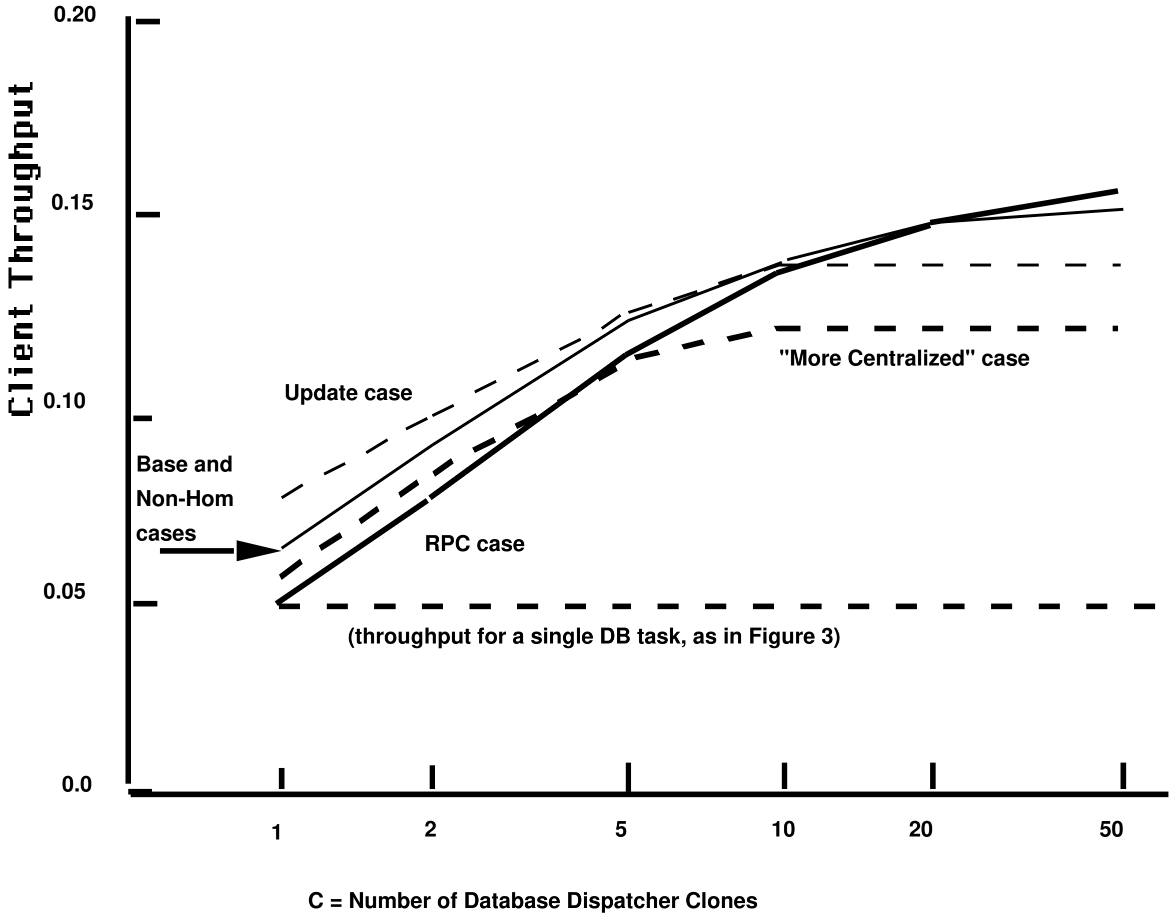

Figure 5 Throughput against Cloning Level: Various Cases

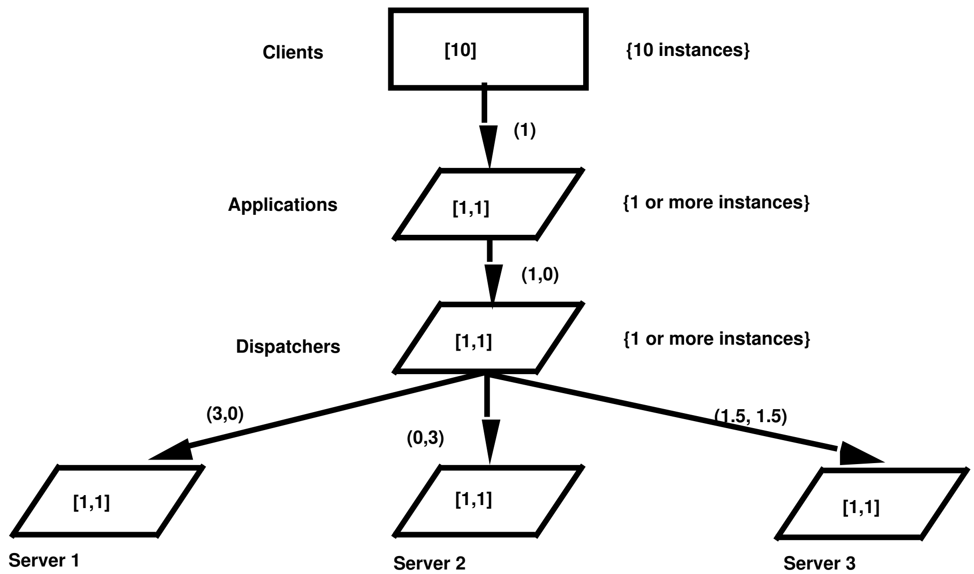

Figure 6 Application Layer with Clones

Table 1. Base Case Behaviour with Cloning Level

Cloning Level C	**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	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

Table 2. Parameters for Variations on the Base Case

Parameter	RPC	Update	Non-Homogeneous	More Centralized
T_1	10	10	10	10
y_{122}	1	1	1	1
s_{21}	1	1.3	1	6
s_{22}	1	0.7	1	2
y_{231}	1.75	1	2	1
y_{232}	1.25	0	0	0
y_{241}	0.50	1	0.2	1
y_{242}	2.50	0	0.8	0
y_{251}	0	1	1.5	1
y_{252}	3	0	1.5	0
s_{31}	2	1.4	1	3
s_{32}	0	4.6	2	1
s_{41}	2	1.4	5	3
s_{42}	0	4.6	1	1
s_{51}	2	1.4	1	3
s_{52}	0	4.6	1	1

Table 3. RPC Case Behaviour with Cloning Level

Cloning Level C	1	2	5	10	20	50
Client Throughput	0.050	0.078	0.116	0.138	0.151	0.160
Dispatcher Cycle	19.9	25.6	42.9	72.6	129	249
Dispatcher Queueing	183	111	64.8	44.0	24.5	5.32
Server 1 Queueing	0	0.60	2.48	5.76	11.4	16.2
Server 2 Queueing	0	0.60	2.49	5.75	12.5	29.0
Server 3 Queueing	0	0.60	2.50	5.76	12.3	31.1
Dispatcher Utilization						
Task	1.0	0.999	0.999	0.999	0.980	0.796
Processor	0.099	0.151	0.227	0.268	0.295	0.311
Server 1-2-3 Utilization	0.300	0.470	0.697	0.825	0.907	0.959
Bottleneck Strength	3.32	2.12	1.43	1.21	1.08	0.83

Table 4. Relationship of Bottleneck Strength to Potential Throughput Increase

Case	Base	RPC	Update	Non-H	"More Central"
Bottleneck Strength at $C = 1$	2.54	3.32	2.09	2.60	2.16
Quick Approximation (Sec. 5)	3.33	3.33	3.33	3.33	2.50
Actual Increase Factor at $C = 50$	2.40	3.20	1.77	2.53	2.14

Table 5. "More Centralized" Case Behaviour with Cloning Level

Cloning Level	1	2	5	10	20	50
Client Throughput	0.058	0.085	0.116	0.124	0.124	0.124
Dispatcher Utilization						
Task	1	0.946	0.874	0.745	0.374	0.150
Processor	0.462	0.682	0.928	0.987	0.987	0.987
Server 1-2-3 Utilization	0.230	0.340	0.464	0.496	0.496	0.496
Bottleneck Strength	2.16	1.39	0.94	0.75	0.38	0.15

Table 6. Client Throughputs vs. Cloning Levels

Applications Dispatchers	1	2	3	4	5	6	7	8	9	10
1	0.063	0.065								0.066
2	0.081	0.092	0.093							
3	0.088	0.104	0.108	0.108						
4	0.092	0.111	0.116	0.119	0.118					
5	0.093	0.115	0.122	0.124	0.126	0.126				
6			0.127	0.129	0.131	0.131				
7			0.129	0.132	0.134	0.136	0.135			
8				0.135	0.137	0.139	0.138			
9					0.136	0.140	0.140	0.140		
10	0.094	0.123	0.134	0.138	0.140	0.142	0.143	0.143	0.143	0.142
11					0.144	0.145	0.145			
20					0.146					0.152
50										0.155