Scalability and performance experiments using synthetic distributed server systems

Distrib. Syst. Engng **3** (1996) 2–8. Printed in the UK

Scalability and performance experiments using synthetic distributed server systems

C M Woodside† and C Schramm

Deptartment of Systems and Computer Engineering, Carleton University, Ottawa, K1S 5B6, Canada

Abstract. The Layered System Generator is used to create synthetic distributed systems of tasks with client–server style (RPC) interactions, representing a wide range of software architectures and workload patterns. A synthetic task system can be used to generate network and workstation traffic which represents the load from a planned software system, so one can observe its probable performance when run on the target network, or its probable impact on other existing applications. It can be used to evaluate the planned software design, or the target network's capability, or both combined. Using LSG, tests were made with systems of up to 39 tasks on a UNIX network, to investigate the performance changes that occur when a small task system is scaled up in size. The performance recorded across the range of experiments was also compared with predictions made by an analytic performance model. The errors were found to be small provided an allowance is made for workstation daemons and similar load components.

1. Distributed server systems

The computing world is moving steadily towards distributed systems running on networks, driven by the need to communicate between applications in different places, by the economics of workstations, and by the opportunity to build reliable systems in this way. Technology to support this is becoming available in the form of remote procedure calls (RPCs), Object Request Brokers such as CORBA [1], and other 'midware' such as security servers. The Distributed Computing Environment (DCE) [2] provides a collection of these features, and the Advanced Network Services Architecture (ANSA) is another such collection.

These distributed processing frameworks support what we will call a 'layered software architecture', with applications in the top layer and service requests descending through the layers, as illustrated in figure 1. Client–server systems have this structure, with the deeper-layered versions being known as 'three-tiered' client–server systems. This notion of layering is not quite the same as the layers of an operating system or a protocol suite, but has many resemblances since lower level servers tend to offer more generic services, for example file service. We will classify systems in section 2 by their breadth, depth and the balance of the workload between the layers. As systems are scaled up typically their breadth increases, with more clients at the 'top', and (perhaps) replication of servers in the middle layers. Sometimes there is an increase in depth also, due to a reorganization of services to divide local services which can be replicated from global services which cannot. This paper uses a narrower idea of scalability, and measures it by the ability to get satisfactory performance from a set of tasks as the breadth is increased. A scalable system is one which can be adapted as the number of users is increased, by replicating servers, to give a proportional increase in throughput (or, to retain the same performance for each user).

There are many performance hazards in distributed systems, including the overhead of the midware, the delays in the security features (obtaining and managing access, encryption), and uncertainty in the delays for obtaining remote services. This makes it worthwhile to study the performance issues while planning the system. Models based on simulation or analytic techniques (for instance [3–5]) may be helpful, but trials on the real network with the real midware are also required. They can demonstrate the achievement of response times under load, and determine the network traffic created to support the application. Some network component which might be ignored in a model—a router, for instance—might turn out to play an unexpected role. A synthetic version of a new system can reveal the amount of overhead in the midware, the effectiveness of priorities in reducing certain important delays, and the sensitivity of the entire system to certain execution times or services, while the software is still in the planning stage. If a proposed new application is to be added to an existing system, a synthetic representation could be loaded together with the existing running applications to determine its impact.

The present study uses a tool called the Layered System

† Email address: cmw@sce.carleton.ca

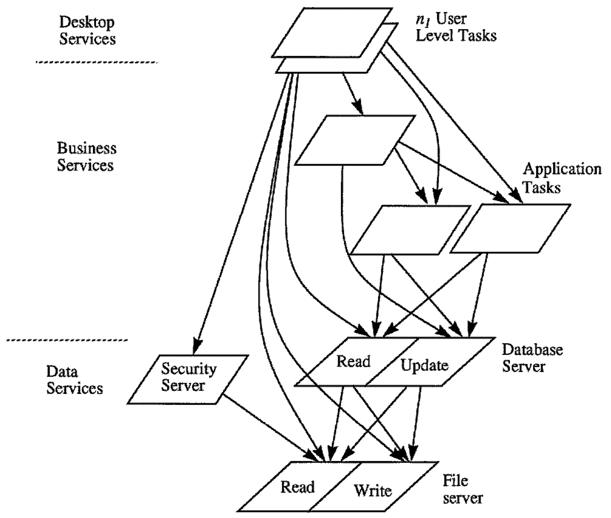

Figure 1. Layered software in a distributed service system.

Generator (LSG) to create a set of tasks which represent the final workload. The tasks execute dummy instructions to create a workload which matches the given execution parameters for each task, and make random choices to simulate their request rates to servers. Systems were created with different depths of layering, and different numbers of clients. They were loaded on a UNIX workstation network and run and measured to show their performance characteristics. The throughput achieved was then compared as the breadth of different layers was increased.

This research borrowed the idea of a synthetic task system from research by El-Rayess, Rolia and MacGillivray [6]. They have described a proprietary 'Performance Modeling and Monitoring Center' (PMMC) tool which also generates synthetic processes in a layered architecture, to obtain measures of processing overhead for the mechanisms proposed to support distributed computing. They measured the overhead of thread management, RPCs and secure (encrypted) communication in DCE, when running a small layered set of tasks on a fully instrumented network. They then proposed to use the data for overhead costs for building analytic models to predict the performance of larger systems, using the layered queueing network models described previously. This raises the question of accuracy of the predictions, which is addressed in the present paper by generating larger systems, and comparing analytic predictions to results.

In principle the overheads determined by the PMMC could be used with LSG, except that PMMC is a proprietary company tool. For this work each overhead component was measured in simpler separate experiments, and these values were used to construct task systems of any size, with any desired total workloads including overheads. The use made of the task systems is quite different in the present paper, being to generate performance tests, to explore scalability and to validate an analytic approximate model as systems are scaled up.

Avritzer and Weyuker [7] have considered the related question of performance testing for distributed systems. They investigated the ability of a planned new processor configuration to carry a known distributed application, by creating a synthetic version of the new system. Their purpose was to get early warning of any problems before porting the application over, and in fact they found that the new configuration would have inadequate performance. To do their tests they created a synthetic workload which mimicked the intended operation mix at the level of operating-system primitives (as measured by the UNIX utility 'sar') and ran it on the new configuration of processors.

The present work goes beyond Avritzer and Weyuker in imitating not only the operation loadings but also the software blocking relationships that influence the rate at which applications can proceed, when software resources are constrained. However they went further in representing the operation workload of a process. They tried to match the mix of a dozen frequent operations (lread, lwrite, fork, exec, iget, msg, sema,...) while up to now LSG only matches the cpu execution time, and the interprocess remote procedure calling. Their workload mix could be used with LSG, to obtain a performance tester with the advantages of both.

It is of considerable interest to know if the performance information could instead have been obtained from an entirely analytic performance model, so a comparison is made between the measured performance and the results of a model, over a range of sizes of systems. Provided an allowance was made for other workloads such as UNIX daemons and logged-in users, the model results were quite close, suggesting that for systems which meet their assumptions, the models can be trusted. Other analytic models, that do not use the layered queueing concepts, have also been proposed for client–server systems. Some of them are clearly toy examples to show off certain model features [8] but others are quite detailed. Ibe and Trivedi [5] described a model using Petri Nets which is in this class, and the book of Menasce *et al* [9] uses queueing network models without layering for this purpose. In both cases the authors only consider user processes with a single layer of servers, and we have shown in previous research that deeper layering makes a substantial difference to the bottlenecks that may occur and the traffic generated by the requests [10]. For this reason we prefer the layered approach to modelling.

The layered queueing model has advantages over standard queueing models and Petri Nets. While Petri Net models can faithfully represent the details and interactions, they suffer from state explosion when they are scaled up to realistic sized applications. Decomposition techniques may eventually make Petri Nets practical. Standard queueing networks do scale up but they leave out important effects of software delays due to request queueing. Layered models basically treat each software server as a queue server and determine waiting for requests. The solution algorithms we have developed for these models use mean value analysis to solve a series of approximate models, layer by layer, as described in [3, 4]. The structure and parameters of a layered queueing model are specified within the concrete architecture, as described in the next section.

The synthetic system has user and server processes that create a simple computational load and a series of interprocess requests, as defined by a special notation. In fact it uses the same notation as the layered queueing network models in [3]. The definition of a system will be called its 'concrete architecture'.

2. Concrete architectures for layered service systems

Each synthetic server system is created from a description of its 'concrete architecture' which defines the actual UNIX processes and their interfaces, and the services each one requests from lower level servers. Processes will be called 'tasks'. One server may offer several different services with a different workload for each one. For instance, a repository may offer a browsing service to view data, and an update service. The update service would generate a very different workload, and perhaps different requests to other servers. In the concrete architecture each distinct service is represented by a distinct 'service entry', usually called an entry, and the workload parameters are associated with the service entries. A task with just one service has one entry, and a top-level client task has been assumed to have one entry. In single-entry tasks the entry can have the same name as the task.

This description of a concrete software architecture is equivalent to a graph with entries represented by nodes and entry-to-entry accesses as arcs, with the entries of a task grouped together in a hyper-node which represents the task. Figure 2 shows the architecture graph for the system in figure 1. Our layered architectures have acyclic graphs, and if they are arranged with the arcs pointing downwards (as in figure 2) then the user-level 'pure client' tasks are at the top, and 'pure servers' (that do not request any lower services) are at the bottom. These are typically system servers like file servers or print servers. If we need to determine the exact layer of a given server it can be taken as the maximum path length from any 'pure client' task, to that server, by following and counting the request arcs on the path.

The above description relates just to the software architecture. The workload parameters and the execution environment also need to be described, to define a full synthetic server system. The additional information is given as parameters of the entities named in the architecture:

Task:

- processor on which it executes (a real processor in the target system)
- priority of the task

Entry:

- average workload parameters (e.g. a CPU time of S seconds) including overheads
- average number $y(e)$ of RPC requests to each lower entry e used by this entry
- for a pure client, a wait time T for thinking between requests.

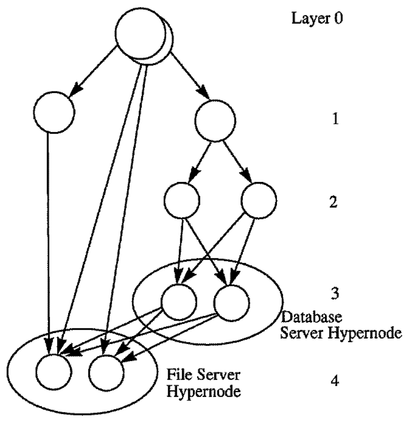

Figure 2. Architecture graph for the system of figure 1.

The workload parameters could include file operations as well as CPU time (as in [6]) or could include system calls as in [7]. In its present form, LSG only defines CPU time parameters. The generalization to include other operations seems to be straightforward. At present the CPU time is calibrated for each processor type, to give the specified number of milliseconds, by straightforward calibration experiments using special instrumented test procedures.

When we are creating a system from task templates, the definitions of the tasks and entries are basically definitions of the templates. However the concrete architecture may include many instances of each template, and it includes a separate task definition for each instance. The structural definition is the same as the template, but the CPU execution parameters depend on the workstation type, and the specific lower service entries accessed will be different for different instances (for example a client may access a particular replica of a server).

2.1. Concrete architecture 'shape'

Software is normally designed not for a single deployment with just one concrete architecture, but for flexibility and a range of deployments. For example distributed database software would be designed for organizations with different numbers of database sites, different numbers of client applications accessing it, and different numbers of users. Flexibility enables the software not only to serve different potential customers, but to allow a given customer organization to evolve freely. Therefore for a given software system one is interested in its performance potential across many concrete architectures and target environments.

In our terms the software design creates a set of task templates (or classes) which can be deployed in different concrete architectures. The possibilities can be summarized in changes to the depth, breadth, and load distribution of the system, which together will be called its shape. We will use the following definitions:

- depth of a system: the number L of layers;
- breadth of a layer: the total number of tasks in the layer. (For layer i it is n_i);

- workload share of a layer (its part of the load distribution): the fraction or percentage of the total CPU work to complete an average response, which is executed in that layer. (For layer i it is x_i.)
- shape of a system: a list of values of breadth and workload share for each layer, from the top down, in the form

$$n_1(x_1)/n_2(x_2)/\ldots/n_L(x_L).$$

The shape is a much simplified summary of a concrete architecture. It ignores important performance determinants like the processor allocation of the tasks, and it ignores the fact that a single layer can contain tasks of very different character. However, we have found it does capture useful comparative information. If one system is scaled up from another, so that they are based on the same tasks and differ only in breadth, this gives a quick summary of the differences over the layers. If there is a shift in workload between layers it exposes bottleneck possibilities at heavily loaded layers. If there is a small change in depth by dividing one task between layers it also captures the extent of the change in the layer workload share. So the shape is useful but it is not intended to be a complete definition of an architecture.

The workload share of a layer is found by first finding the total workload per user-level response, for each task in the layer, then adding up the work over the layer, and expressing it as a fraction of the sum over all layers. To find the workload of each task for one type of response, one traces down through the layers to find how often each entry is invoked, multiplies by the execution time S for the entry, and adds up over the entries. If there are several types of response one must know the relative frequency of each type, and combine the workload figures in proportion to their relative frequency.

An example of the calculation of workload shares will be given with the experiments for case 5, in section 6.

3. The synthetic server system

The synthetic system is made up of instantiations of a single generalized task template, with computation parameters that impose the desired workload quantities for each given concrete task. The parameters of each entry are read from a file during initialization. The parameters are: S the total execution time of the service; $y(e)$ the average number of service requests made to lower entry e (sum of $y(e) = Y$); s the average execution time for a 'slice' of execution by the entry, between requests for lower-level service, with an extra slice at the end, $s = S/(Y+1)$; $p(e)$ the probability that a slice ends with a request to lower entry e, $p(e) = y(e)/(Y+1)$; $p(END)$ the probability the slice ends the service, $p(END) = 1/(Y+1)$.

The behaviour of the task template can then be described as follows:

```
loop forever
    accept a new request for entry $E$
    while (not $END$)
        • generate a time $t$ with average $s$ and expend time
          $t$ by a calibrated repetitive operation
        • generate an entry $e$ or the condition $END$, with
          probability $p(e)$ or $p(END)$ respectively
        • if not ($END$) make a service request to entry $e$,
          by RPC
    end while
    send RPC reply to requester
end loop
```

For a 'pure client' top-level task there are no requests arriving, so the flow is simpler. The loop simply starts again after completing, without waiting for a request to arrive. Also the loop begins with an optional waiting time with mean T, to represent the time a human user might spend between inputs. Also,

- file I/O can be included by having certain entries designated as file service entries, and having them generate file I/O requests instead of RPCs to servers.
- we could use a synthetic spin loop representing the planned application code, instead of the total execution. In LSG, a calibrated server 'slice' time was determined to give the desired value of s for a 'slice', including overheads such as for the RPC.

3.1. Loading and running an experiment

The set of processes needs to be loaded and run on the target network. For this one could use a script, or a system like PVM. We used a program called DECALS [11] that was previously developed to run performance experiments in a distributed heterogeneous enviroment. DECALS provides facilities for the automatic launching of experiment sets, including synchronized startup and shutdown, and initialization of the task templates via parameter files. As well, we used DECALS's event monitoring facilities to capture events through software probes in the task templates, to determine response time and user-level throughput.

4. Scaling experiments on systems with one server layer

The first experiments were done on a system with a set of user tasks and a single server, as shown in figure 3. It is a simplified version of the most common concept of a client–server system. In reality the server is usually a database system, possibly on a mainframe computer; here we have represented it by a single process with only a CPU load. Nonetheless this task could approximate a real server with the given CPU load, and in which the storage subsystem is not a bottleneck.

The simplest scaling issue to examine is the capacity of the server to handle an increased population of user tasks, with no change at the server. As the population increases the server will saturate and response times will begin to increase due to waiting in the server queue. Then we could look at expanding the second layer to n_2 servers, giving the shape

$$n_1(.)/n_2(.).$$

We might hypothesize that we will need one server for every N users for some value of N; our search then is to determine N and see what it depends on. Again a simple hypothesis is that, if the server work is t_2 seconds per response and a user can generate a new request t_1 seconds after a previous response, then N is roughly t_1/t_2. If users run desktop devices that share workstations this could be modified. We studied a system in which each response is 90% executed by the user tasks and 10% executed by the server, thereby giving a shape:

$$\text{case 1} : n_1(0.9)/1(0.1) \quad \text{for } n_1 = 1, \ldots, 10.$$

In the actual experiments we set the parameters of the user tasks at $S = 135$ ms and $T = 0$ (to generate as much load as possible from a limited number of user tasks), the service requests at one per user response, and for the server, $S = 15$ ms. This makes $t_1 = 0.135$, $t_2 = 0.015$ and our estimate of $N = 9$. As stated previously the service times were calibrated to include the RPC time to marshal and unmarshal messages. The message sizes were all set to messages of 16 bytes. The experiments used various types of station, all running the Solaris version of UNIX system V. Ten replications were run for each set of parameters, with a minimum of 1000 responses for each user task in each replication, and confidence intervals were calculated across the replications.

To consider a system with a similar stress on the server, but with the load shared among three servers, a second set of experiments had 87.5% of a response executed at the user and 12.5% at the server, thus the shape:

$$\text{case 2} : n_1(0.875)/3(0.125) \quad \text{for } n_1 = 1, \ldots, 10.$$

In this case each client had $S = 315$ ms and accessed all three servers an average of one time each, per response. The servers had $S = 15$ ms as before.

The results for the total user throughput give a good idea of scalability; in a fully scalable system the total throughput is proportional to the number of users. From the 20 experiments defined above, the results are shown with whiskers for the 95% confidence intervals attached in figure 4. As we might expect, the throughput initially rises linearly with the users and then flattens out as the server saturates. The saturation effect is evident well below nine users, which might have been predicted above. In this case it is due to the client tasks sharing just three workstations. In the cases with three servers the corresponding level is $3 \times (0.875/0.125) = 21$ users, and again the saturation is visible well below this level.

The saturation above three user tasks turns out to be due to saturation of the three processors running the user tasks, which are computationally intensive. The server utilization levels off at about 30% in case 1 and about 12% in case 2. The case 2 throughputs are lower because each response does three times as much work, however case 2 achieves more than one third the total throughput of case 1, because of greater parallelism at the server level.

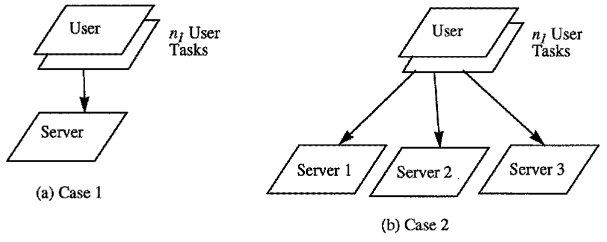

Figure 3. Systems with one layer of service.

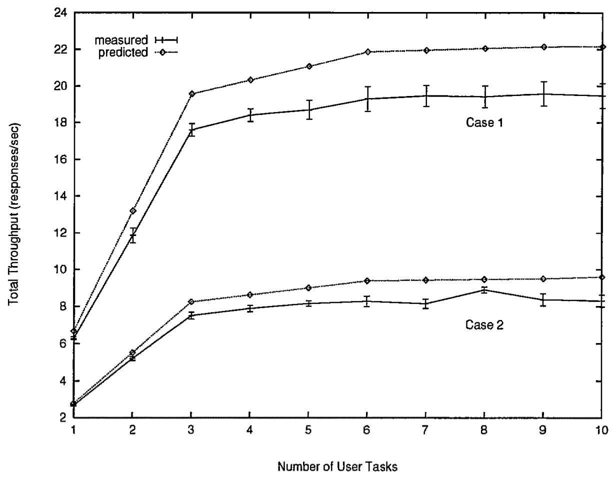

Figure 4. Systems 1 and 2.

The analytic model accurately reflected the software architecture and the allocation to processors. The results are close to the measurements and accurately reflect the trends. However the results are consistently higher than the measured throughputs by about 12–15%. This is because the analytic model was not corrected for the additional workload present on each workstation during the experiment. This additional workload has three components:

- UNIX daemons which are estimated at about a 2% load on each processor;
- DECALS overhead, which is roughly 2%;
- other users, doing word processing mostly.

None of these loads is easy to estimate precisely, but the relationship between the prediction and measurement is very consistent. We conclude that we can trust the analytic model, with the caveat that an allowance must always be made for competing workloads. If the competing workload utilization is known the analytic model can be adjusted to account for it.

5. Experiments with two server layers

Instead of just replicating the servers in the second layer we may also be able to divide each service into a 'front-end' part and a 'back-end' part, with the latter implemented in a deeper layer of servers. This divides the service load further

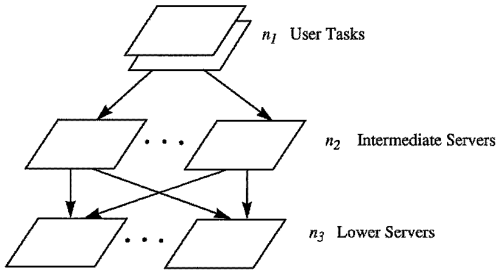

Figure 5. Systems with two layers of service (cases 3 and 4).

and increases the basic capacity of the system. Figure 5 shows a second layer of servers. Experiments were done with two shapes:

$$\text{case 3} : n_1(0.8)/1(0.1)/1(0.1) \quad \text{for } n_1 = 1 \text{ to } 10$$
$$\text{case 4} : n_1(0.5)/3(0.3)/2(0.2) \quad \text{for } n_1 = 1 \text{ to } 10.$$

Case 3 can be compared with the single layer experiments in the previous section, since each server takes the same share (10%) of the total load. Case 4 has a fatter service portion and a smaller total client share of the work, which might characterize systems needing deeper service.

Seven workstations were used for the experiments. The client tasks were distributed as equally as possible among three workstations, with servers from the same layer on different workstations. Replications for confidence intervals were used as before.

The total throughput results are shown in figure 6, with whiskers for the 95% confidence intervals. The dotted line again represents the predictions by an analytic layered queueing model. We might expect case 3 to be worse than case 1, because each request has two opportunities to queue, but this is offset by the fact that more of each request is done by the servers. Again the analytic model predictions are a little higher than the reality, by much the same ratio as in cases 1 and 2.

6. Experiments on deeply layered systems

Scalability is less obvious in deeply layered systems, where saturation may be introduced in many different ways. Two cases with five layers were considered to explore this effect, with 19 tasks in case 5 and 39 tasks in case 6, and differing degrees of replication of servers. These might represent a three-tiered client–server system as described by Febish and Sama [12], with a user services tier at the top level, a data services tier at and near the bottom, and a business services tier, perhaps with many layers, in between. The business services tier provides composite services. The shapes of the two systems are:

$$\text{case 5}: \ 10(0.8)/3(0.05)/3(0.05)/3(0.05)/3(0.05), \ 19 \text{ tasks}$$
$$\text{case 6}: \ 20(0.82)/10(0.1)/5(0.05)/2(0.02)/1(0.01), \quad 39 \text{ tasks}$$

and figure 7 shows the concrete architecture in both cases. They represent different approaches to dividing the work

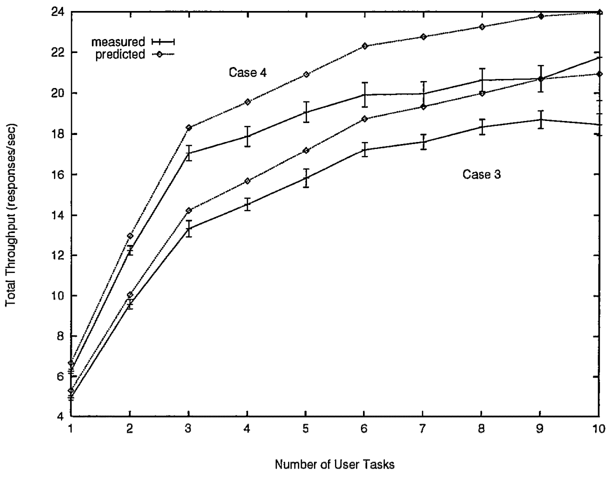

Figure 6. Throughput results for cases 3 and 4.

between layers. In both cases, each task below the top layer takes the same share of the work per response (1% in case 6, 1.67% in case 5). However case 5 has equal balance among the service layers, while case 6 has a tapered workload with more servers at the higher layers, to share a larger percentage of the load. Case 6 has just about twice as many servers to serve twice as many top-level clients; the question is, can it provide twice the throughput?

Case 5 will provide an example of the calculation of a *load distribution*, as promised above. The execution times of the user tasks are $S = 2304$ ms, and for the tasks in successive layers down from there, S is 72, 36, 18 and 9 ms respectively. Each task requires, besides its own execution, one request to each task on the next layer down. Thus in a typical response there is one invocation in the user layer, 2 in the next layer, and then 4, 8 and 16 invocations in the lower layers, in order. Each layer below the top thus contributes 144 ms of execution to a response, giving a total of 2880 ms in total for a response. The share of the top layer is $x_1 = 2304/2880 = 0.8$; of all the lower layers it is $x_i = 144/2880 = 0.05$, as defined for the case.

The results with 95% confidence intervals were:

$$\text{case 5}: \ \text{measured } 1.18 \pm 0.045, \ \text{analytic prediction } 1.24$$
$$\text{case 6}: \ \text{measured } 3.545 \pm 0.142, \ \text{analytic prediction } 3.656.$$

Case 6 with its tapered shape has provided more than twice the throughput (for twice the users), compared to case 5, showing that workload should be located as high as possible in a deeply layered architecture, and should be supported by replication. The accuracy of the analytic predictions for these two cases is excellent.

7. Conclusions

The value of doing real network tests with a full-scale synthetic system is (1) that they verify any assumptions that have been made about device and network capacity, and system overheads and (2) that they reveal any performance-limiting component in the network. In cases 1 and 2 of this paper, for example, the performance-limiting factor was not

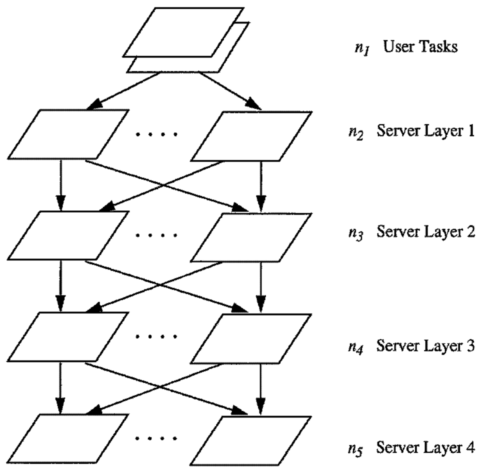

Figure 7. Deeply layered systems (cases 5 and 6).

the servers but the processor support for the user tasks. This was not totally unexpected in these cases, but it illustrates the point.

The analytic modelling by the LQNS solver gave quite accurate predictions across the full range of systems examined here, provided allowance is made for competing workloads that have not been modelled. For example, in cases 1 to 4 the synthetic user tasks were run on workstations that were also in use for text editing, program debugging and e-mail. As these workstations were the saturation point in the system, the entire throughput in the saturated range is reduced by the fraction of cpu that was devoted to the competing work—about 12% to 15%.

For cases 5 and 6 the accuracy of the analytic predictions is very good—5% and 3% error, respectively. These tests were run in a relatively quiet period, with smaller competing workloads. Thus, for most practical purposes the analytic predictions are accurate enough, and accurately reflect performance trends. Some care must always be taken to account for potential effects of competing workloads. If they are well known, they can even be included in the analytic model itself.

In summary, LSG seems to fill its desired role for performance testing, and the results also back up the use of analytic modelling. Further development of LSG to generate multithreaded server tasks, and extensions to the workload characterization to include file operations and other kernel operations, are underway.

Acknowledgments

This research was supported by grants from TRIO, the Telecommunications Research Institute of Ontario, and by NSERC, the Natural Sciences and Engineering Research Council of Canada. We wish to thank Alex Hubbard, who developed DECALS and helped us to use it with LSG; Greg Franks, who helped program the definition of the synthetic system, and Jerry Rolia who suggested synthetic task systems to us in the first place.

References

[1] 1992 Object Management Group and X/Open *The Common Object Request Broker: Architecture and Specification* Framingham, MA, USA and Reading, Berkshire UK

[2] Open Software Foundation 1992 *Introduction to OSF DCE* (Englewood Cliffs, NJ: Prentice Hall)

[3] Woodside C M, Neilson J E, Petriu D C and Majumdar S 1995 the stochastic rendezvous 15 network model for performance of synchronous client–server-like distributed software *IEEE Trans. Comput.* **44** 20–34

[4] Rolia J A and Sevcik K C 1995 The method of layers *IEEE Trans. Software Eng.* **21** 689–700

[5] Ibe O C, Choi H and Trivedi K S 1993 Performance evaluation of client–server systems *Parallel Distrib. Syst.* **4** 1217–29

[6] El Rayess A, Rolia J A and MacGillivray R 1995 Performance prediction of distributed applications using the performance modeling and monitoring center (PMMC) *Proc. 6th IEEE Int. Workshop on Distributed Systems Operation and Management (Ottawa)*

[7] Avritzer A and Weyuker E J 1996 Deriving workloads for performance testing *Software Pract. Exper.* to appear

[8] Buchholz P 1993 Aggregation and reduction techniques for hierarchical GCSPNs *Proc. 5th Int. Workshop on Petri Nets and Performance Models (Toulouse)* (Los Alamitos, CA: IEEE Computer Society Press) pp 216–25

[9] Menasce D A, Almeida V A F and Dowdy L W *Capacity planning and performance modeling* (Englewood Cliffs, NJ: Prentice Hall)

[10] Neilson J E, Woodside C M, Petriu D C and Majumdar S 1995 Software bottlenecking in client–server systems and rendezvous networks *IEEE Trans. Software Eng.* **21** 776–82

[11] Hubbard A, Woodside C M and Schramm C 1995 DECALS: Distributed Experiment Control and Logging System *Proc. CASCON'95, Meeting of Minds (Toronto)* (Toronto: IBM Center for Advanced Studies) pp 146–60

[12] Febish G J and Sarna D E Y 1995 Building three-tier client–server business solutions *White paper* (Englewood, NJ: ObjectSoft Corp.)