# Automatic Generation of Performance Models for Distributed Application Systems

M. Qin, R. Lee, A. El Rayess, V. Vetland, and J. Rolia

**abstract**

Organizations have become increasingly dependent on computing systems to achieve their business goals. The performance of these systems in terms of response times and cost has a major impact on their effectiveness. To achieve openness and scalability, these systems have begun to rely on distributed environment technologies such as the Open Software Foundation's Distributed Computing Environment (DCE) and the Object Management Group's Common Object Request Broker Architecture (CORBA). The resulting systems are complex and have performance behaviour that can defy intuition. Predictive performance models and performance evaluation techniques are needed to help manage their behaviour. Predictive models allow the timely evaluation of the performance impact of many different application loads and alternative configurations. Unfortunately model building by hand can be time- consuming and error-prone. This paper describes techniques developed to help automate the construction of performance models for operational distributed application systems. It describes the information that is needed to automate model building, the infrastructure that supports the model building, and the methods used to obtain or estimate the required information. An example is given to illustrate the model-building process and the evaluation of several "what-if" scenarios for management that can be answered using

the model.

Keywords: Distributed applications, performance management, performance evaluation

# 1 Introduction

Distributed computer systems enable the sharing of an organization's information and computing resources. Response times experienced by end-users are typical measures of quality of service for such systems. If the response times of an organization's system are too long then its operational effectiveness will be hampered. Tools and techniques for managing the performance of such environments are essential but are currently not available.

Presently, key applications are often supported on centralized subsystems. If performance problems occur, experts familiar with a subsystem can improve its behaviour using subsystem-tuning techniques. For example, changes can be made to affect operating system, network, or database behaviour, or to speed up existing resources or add a small number of resources.

To scale upwards towards an organizational or global processing model, new and reengineered applications distribute their functionality across heterogeneous distributed computing facilities. Though *system management* from the perspective of managing individual subsystems will always be important, this strategy does not sufficiently address distributed application system performance problems. Instead, we must approach the problem from the perspective of the applications themselves. Performance-oriented design and management shifts, from understanding a collection of independent centralized subsystems to understanding the correlations between distributed application components and the loads they place on the computing facilities available to them. Currently available measurement packages do not provide sufficient information to support such management in heterogeneous systems.

A collection of programs in execution that cooperate to accomplish some overall objective is called a *distributed application*. Instances of programs correspond to operating system processes. Performance management from the perspective of the applications is

called *distributed application performance management*. A performance model for a system of applications can be used to detect performance bottlenecks and to conduct "what-if" analysis for exploring alternative system configurations.

Performance modelling methodologies, instrumentation, and tool support exist for centralized systems. In such systems, the contention for hardware is prominent. Workloads are often characterized in terms of users or groups of users and their use of a relatively small number of hardware resources. For distributed application systems, the picture is quite different. Software servers can cause queueing delays and may become bottlenecks. Thus they must be represented explicitly in predictive models along with a characterization of their use of hardware. We are forced to include this greater detail in our performance models. Unfortunately, doing so without propper measurement system and tool support makes model-building time-consuming and costly.

For operational decision-making, building models for distributed application systems “by hand” is impractical. New instrumentation and tools are needed to help support and automate model building. Fortunately, there is a trend towards the standardization of distributed system technologies. Most vendors are working towards "open" systems, in which heterogeneity is possible because of standardized protocols and interfaces. This trend makes it more feasible to include support for the new instrumentation needed for distributed application performance management.

This paper presents a cost-effective method for performance model building that enables the performance management of distributed applications. The structure of the paper is as follows. Section 2 describes the informational requirements for model building and our measurement infrastructure. An instrumentation package that collects data about applications is documented in Section 3. Section 4 describes a server for performance data and how it is used to estimate some values that are difficult to measure directly, but nevertheless are needed for predictive models of distributed applications. Section 5 presents the steps of performance model generation. An example is given in Section 6. Section 6 presents some practical experiences and concluding remarks.

# 2 Information, models, and infrastructure

In this section, we establish terminology, describe the information that is needed for model building, and the infrastructure that supports such model building. We give a high-level overview of the model-building system. In subsequent sections, more detailed descriptions are given of the infrastructure's key components.

Each program that runs as part of a distributed application has zero or more *interfaces* that define the services it provides (if any) to other programs. The *services* are also called *operations* or *methods*. Interfaces are typically defined, for example in DCE and CORBA, using an interface definition language (IDL). The IDL for an interface looks much like a typical include file (".h") for the C language. Processes *export* interfaces that they support and *import* interfaces that they use. Location transparency in distributed systems is achieved using a name server. In DCE this is referred to as the Cell Directory Service, and in CORBA it is the Object Request Broker. A name server permits connections to be established at the time of execution between (client) processes and the interfaces of serving processes. Automated model building must keep track of these connections.

The predictive model we use is called a Layered Queueing Model (LQM) [1, 2, 3]. Layered queueing models include a characterization of the operating system processes that contribute to a system, their relationships, and their use of hardware resources. The connections between processes, "visit information" between operations of interfaces, and resource demands by operations on hardware resources must be determined. These are found either by measurement or by estimation based on measurement data. The parameters for an LQM are defined in more detail in Section 3. Once the model has been defined, performance evaluation techniques are used to estimate mean response times for system users. The response times include estimated queueing delays caused by contention for shared resources such as software servers and hardware.

Many performance evaluation techniques are available. Some are analytic, like Mean value analysis [4] and Petri nets [5], while others are based on discrete event simulation. Analytic techniques can usually provide results orders-of-magnitude faster than simulations.

The Method of Layers (MOL) is the analytic technique that we use to predict the behaviour of the LQMs we generate [1, 2]. Petri-nets are not used because, with today's technologies, their solution time for this class of models is even greater than that of simulation. The LQMs we construct can also be simulated but only include the detail of the analytic model. Simulation models can represent arbitrary detail within a system, but the time it takes to develop a detailed model and to run it may be prohibitive. The simulation may need to run for a very long time to obtain statistical significance if a large number of requests is modelled and the system has a highly utilized resource, or if there are significantly different time scales for parameters in the model.

The capture of parameters for performance models is in itself an important issue. The additional instrumentation needed to collect parameters for LQMs must not introduce too much overhead into an operational system. The monitoring strategy we use is based on that of the Distributed Measurement System [6]. We need to add instrumentation code to the programs so their corresponding processes report the data needed for model building. Each process periodically reports a summary of its metrics to a performance data server. Typical reporting intervals may be in the range of minutes. It has been shown [7] that monitoring overheads with this paradigm can be expected to introduce an overhead for most monitoring scenarios below several percent for both processor and network loads.

Unfortunately, there are some model parameters that we require but that cannot presently be measured directly. Statistical techniques have been proposed [8] for estimating these values, based on measured data for several periods. The performance data server supports the techniques described in Section 4.2. The model-building tool makes requests to the performance data server to gather information needed to generate the LQM automatically. The MOL can then be used to solve the LQM in order to estimate contention delays.

The model building system has three main components:

- An instrumentation package that collects measurement data from application processes and forwards the data to a performance data server;
- A performance data server that stores captured data and estimates model parameters

that cannot be measured directly;

- A model builder that gathers information from the performance data server and constructs the LQM.

In the following sections we describe each of these components in more detail.

# 3 The application program instrumentation package

The purpose of the instrumentation package is to measure and collect the information required for model building and forward it to the performance data server. The package includes a variety of routines and macros to be placed by an application programmer at certain points within the application's code. Though this solution is not ideal, it has enabled us to gather further information easily about software interactions that are needed to test our model building approach. The instrumentation we describe should eventually be integrated with midware runtime systems, code generators (such IDL compilers), or communication class libraries (such as the Adaptive Communication Environment ACE [9]). This approach hides instrumentation details from application programmers and provides more information about events in the midware.

In order to construct a predictive model of an operational system, it is necessary to collect information about the system's configuration and its resource consumption. We need to identify the applications that currently consume resources, the processes that participate in each application, and the connections between these processes. Furthermore, we need to capture information about the operations of processes, the visit counts between them, and the amount of physical resources used by each operation.

The above metrics describe the parameters of LQMs. The LQMs describe contention for physical resources and for software servers. Performance evaluation techniques such as the MOL [1, 2], and simulation can be used to predict the contention effects of the modelled systems. LQMs have the following parameters:

- Identifiers, threading levels, and think times of operating system processes;

- Scheduling policies of devices;
- For each method ($m$) of each interface ($i$) of each process ($c$):
  - the average number of visits ($V_{c,i,m,k}$) to each device ($k$) in the network;
  - the average service time ($S_{c,i,m,k}$) per request at each device ($k$) in the network;
  - and the average number of visits ($V_{c,i,m,s,i_2,m_2}$) to each method ($m_2$) of each interface ($i_2$) of each of its server processes ($s$).

The *think time* of a client process is the average time it spends waiting for input from users. Note that the service times at the methods of serving processes are not specified. These values are a function of the dynamic system state and must be estimated by performance evaluation techniques.

We now consider the behaviour of the instrumentation. During the start-up period, an instrumented process invokes a routine called `observer_init()` to instantiate and fork an observer thread. The observer thread communicates with the underlying operating system to obtain information that identifies the instantiating process (called the process ID) and the node on which it is executing (called the host internet protocol address).

The observer thread has a life-cycle of sleep-measure-report. To manage observation overhead, the observer reports measured data to the performance data server with controllable frequency. The observer thread "sleeps" for an interval specified through the observer_init() routine. After it “wakes," it measures the physical resource usage of the process and gathers metrics collected for the operations of interfaces since the previous report, and sends them to the performance data server.

Examples of physical resource requirements include the CPU time, physical read and write counts, and the network communication bandwidth used. The data collected about operations is event-based and contributes to running sums for visit counts and response times. At the time to report, the observer aggregates these measures for the interval and reports them. In this way the reported data is *intervalized* and reported. Note that this is

not a pure sampling technique in which the current state of the process is simply observed at the instant of reporting. Important information would be lost with such an approach.

Ideally, hardware resource consumption values would be available for each operation of each interface in each process. Unfortunately, physical resource demands are usually reported by operating systems on a per-process basis. For these systems, statistical techniques can be used to help estimate the measures on a per-operation basis. This is described further in Section 4.2. Operating systems with kernel-based threads typically report measures for each thread separately, and may in the future be better able to support the characterization of physical resource consumption at the operation level.

The information needed to characterize operations of interfaces is collected by instrumentation macros in the program code. A block of code, representing an operation, that needs to be measured, is surrounded by a pair of macros that updates the visit counts and measures response times for the operation. The instrumentation macros also capture the relationships and visit counts between the operations of process interfaces as follows.

Each process ($c$) exports a set of interfaces ($E_c$) with their protocol sequences, and imports a set of interfaces ($I_c$) with their protocol sequences. Processes that do not export any interfaces are pure clients. We assume that they are characterized by one virtual interface. A block of code that implements an exported operation of an interface is called an input operation (the name is derived from the process's own point of view), and is surrounded by the following pair of macros:

```
S_IN_OP /* Start INput OPeration */
E_IN_OP /* End INput OPeration */
```

Similarly, a block of code that defines an imported operation of an interface is called an output operation and is surrounded by the following pair of macros:

```
S_OUT_OP /* Start OUTput OPeration */
E_OUT_OP /* End OUTput OPeration */
```

The macros characterize operation invocation counts and response times. The pairs also interact with the operating system and/or midware to obtain information about which processes are connected. This information is discovered by exploiting imported and exported interface protocol sequence information.

Borrowing from DCE terminology, a *protocol sequence* is a combination of transport protocol identifier (TCP/IP), node address (an IP address), and a port (socket or endpoint) identifier for an interface. For a DCE application, this information can be obtained by querying the binding handles of each imported and exported interface.

All information collected by the macros is stored in a data buffer that is shared among the threads of the process. When the observer thread "wakes," it intervalizes the measurement data stored in the buffer and sends it to the performance data server.

# 4 The performance data server

The performance data server supports storage for the intervalized process data gathered by the process instrumentation. The data server also performs data reduction and some statistical analysis needed to derive parameters for LQMs. In the following subsections we describe the services provided by the server and how they are used.

## 4.1 The Interface

The performance data server provides an interface that supports the storage, analysis, and reporting of performance data for monitoring and model building. It is a multi-threaded server that supports more than one access method. The interface can be accessed via a TCP/IP socket interface or a DCE Remote Procedure Call (RPC) interface. The two access methods are provided to enable the data server to operate in heterogeneous environments. The performance data server has at least two threads, one to handle DCE RPC, and the other to handle requests via a TCP/IP socket interface. A default well-known port number is used to create a socket. Clients use this port number and the host name of the performance data server to connect to the performance data server. Currently the socket interface is only able

to support one connection at a time, and hence one client at a time. At present this is not a severe limitation since the observer threads and the model builder use the DCE interface, which supports multiple callers. Access to the data server's files are protected by mutual exclusion.

The performance data server must be running before data gathering can begin. It provides ten services to clients. They can be categorized into three types of services: management, data storage, and data reporting.

**Management** Management services are used to initialize data files, to delete data, and to remove the performance data server from the system. They are invoked by management applications only, *not* by managed applications. The `init()` service deletes all existing performance data. If the data files do not exist, the files will be created. `del_period()` deletes all intervalized data in the file belonging to a specified process for a specific period of time. `del_all()` is similiar to `del_period()`, but applies to all processes of all applications. `shutdown()` stops the server but leaves the data file and supporting files intact. `uninstall()` removes the data file and any supporting files.

**Data storage** The data storage service permits observers to store measurement data. Observers interact with the performance data server using the `put()` service. It enables observers to store the collected data in files. The data can then be obtained by management applications using the data-reporting services supported by the performance data server. Examples of stored data include: the time interval for the reported data, application, process, and interface identification information, and metrics for operations of imported and exported interfaces.

**Data reporting** Reporting services provide data for management applications. The services all take a time period as input. The `get_applications()` service returns a list of identifiers for applications that were active within the specified time period. `get_host_process_ids()` takes as input an application identifier and returns the

list of application processes that reported data during the measurement period. The information about each process includes its host name, process ID and program ID. This information is needed as input to the `get_observation()` and `get()` functions. `get_observation()` takes an initial time as one of its arguments and returns data for the first reported interval for a process that starts after the argument. This service can be used to acquire data for display, or to discover the time at which a process first starts to report. `get()` takes as input a description for a process and a time period. It then aggregates information for the process over the time period, computes confidence intervals, and performs further statistical analysis, as described in the next subsection. The data returned is an aggregate of the intervalized data submitted to the data server by the `put` operation.

## 4.2 Data analysis

The collected measurement data is intervalized by the observer within each process and then forwarded to the data server for storage. The `get()` routine, which is typically called by display or model- building applications, aggregates data over a longer time period that spans many reporting intervals. It provides confidence intervals for the measured data.

We note that LQMs require CPU and disk input-output counts for each operation of each interface exported by a server. Data at this level of detail is needed when services provided by application servers require significantly different numbers or types of resources. Unfortunately, this data is not available on most UNIX platforms. Most UNIX platforms report resource consumption on a per-process basis. Sometimes, the resource demand per operation can be estimated using statistical regression techniques [8].

Consider a process that provides $n$ operations with data collected over $m$ periods. Let the number of invocations for each operation $i$ in the $j$th period be $c_{ij}$ and let $r_j$ be the CPU usage (or disk input or output operations) observed during period $j$. The data forms a set of linear equations as follows:

$$c_{1j}x_1 + c_{2j}x_2 + ... + c_{nj}x_n = r_j, 1 \leq j \leq m$$

The solution to this system of linear equations gives the estimated CPU usage (or disk

input or output operations) of each operation. That is, if the solution is $(x_1, x_2, \cdots, x_n)$, then $x_i (1 \leq i \leq n)$ is the estimated CPU usage of the $i$th operation.

The statistical method presented above has some special cases that are addressed by the performance data server. The number of observations must be greater than or equal to the number of services provided by the given process. If there are not enough observations to perform the regression, the average resource demand for the process per operation is returned for each operation. Another scenario is that the visit counts for several operations may be highly correlated. As an extreme case, visit counts for one operation may always be an exact multiple of the counts for another operation. Regression estimates for strongly correlated operations must be combined. And, if an operation is never invoked, the operation is not included in the regression model. Its resource demands are each set to zero.

# 5 Building the LQM

Within our modelling framework, a predictive LQM is built for a specific time period. The model characterizes the mean behaviour of the system for that period. A typical period may be between 15 and 30 minutes. As a goal, it is long enough to give statistical confidence in the measurements yet short enough to describe an interesting part of a system's day, such as a peak busy period. Once data is gathered for a period by our instrumentation and placed in the performance store, the model builder can be used to generate a corresponding LQM. In this section we consider the purpose of the model builder, as well as verification and validation steps for the model. Section 6 shows how the resulting model can be used to explore "what-if" scenarios.

The purpose of the *model builder* is to acquire measured data from the performance store and construct an LQM that reflects the behaviour of the measured system. The LQM has two main aspects, 1) structure and 2) resource-demand parameters. The structure describes requesting and blocking relationships among processes. The parameters fix the mean number of visits and service times of requests as they flow through various operations of process interfaces and use devices. We assume that interactions between processes

are synchronous RPC. Other interactions and support for them are discussed further in Section 5.3.

We now consider the steps needed to build an LQM using measured data, model verification, model validation, and future work.

## 5.1 Model-building steps

In an LQM, server processes and devices are represented as *entities*. Each entity has one or more *entries*. An entry is used to characterize the visits made by an operation of an interface to other operations of interfaces and devices. In the model, each device has a single entry that is implied. Each entry has a list of *events*. Each event represents the mean number of visits and service times at a device, for example a CPU, disk, or network, or the mean number of visits to an operation of an interface in some other process. In the model, these are reflected as a visit from the entry of the client entity to an entry of the server entity. Devices have a single entry with no events.

To build the model, we must obtain information about the system of applications from the performance data server and then conduct the following sequence of steps:

1. Choose the measurement time period;
2. Get data about the application processes from the performance store for the time period;
3. Associate processes and devices with entities in the model and assign their populations and scheduling disciplines;
4. Create events to reflect interactions between entries of entities;
5. Fix the mean number of visits and service times for events;
6. Fix the think times for entities.

For model building we first choose a time period of interest. As an example, a bank's automated teller machine (ATM) system may be most busy during the half hour from noon

to 12:30 pm each day. A model should be built for this time period. It can then be used for operational performance management. For example, to study the impact of increased loads or alternative application or system configuration on user response times.

The performance data server would have to be started before noon, and the ATM application instrumentation enabled sometime shortly before noon. During the time period, observers within application processes periodically submit intervalized data to the data server. The remaining steps of model building can be completed at any time. For this example, we would choose a time shortly after 12:30 pm.

When model building occurs, the model builder queries the performance data server to get a list of applications that were active during the measurement time and the identities of the processes that participated in the applications. Subsequent queries are made to get data for each process.

Each process is identified by an internet protocol address for its node and a process identifier. In the LQM, entities are created to represent each unique process and the CPUs and disks of each node. If multiple client processes from the same node have the same program type ID, then they are represented as a single entity with a population greater than one. The events of an entity's entries reflect the average behaviour of an entity's corresponding process or processes. For server processes, the population of the entity is set to the process's threading level. Entities are then assigned scheduling disciplines. Devices are assigned the processor-sharing discipline as a default. Processes are first-come-first-served servers.

From the protocol sequence information, we can establish which process interfaces are connected to one another. For each process, each operation of each exported interface is associated with an entry in its corresponding entity. Using count information for visits between operations of process interfaces, we fix the average number of visits to each imported operation (or server entity entry) from each exported operation (or client entity entry). Physical resource demands for the operation (client entity entry) include the average demands of the operation at the processor and visits to the disk of the entity's node. A

system configuration file is used to specify the number of processors and disks on each node and their service rates. Future performance management systems should make such information automatically available for each node.

The think times for client process entities represent the average time between client requests. Think times are reflected in the model as events to a special think centre device. Each entry of each client entity has an event for a single visit to the think centre. The average think time is calculated as follows: take the difference between the duration of the measurement period and process busy times, then divide by the number of process entry completions.

## 5.2 Exceptions, model verification, and model validation

Several problems can arise when relying on an automated model-building system. The model-building system is designed to recognize and overcome some of these problems.

Separate information about each entry's physical resource demands is not available if the regression performed by the performance data server fails. In this case, the `get()` calls to the data server simply return the average physical resource demands per completion at the process. These are used for each entry. The remaining visits between operations of interfaces are still characterized correctly.

If the confidence intervals for data reported by the performance data server are poor, a warning message is given. Either not enough measurement data was collected to characterize the system's behaviour, or some significant change to the system took place during the measurement period. Futher exploratory work can then be done automatically to determine whether all of the processes were active for the entire measurement period. If not, the resulting information can be used to partition the measurement period into several shorter measurement periods and give models for each. This is a subject of future work.

The instrumentation package reports response times for operations of interfaces. These can be used for display purposes and also for model validation. The performance data server reports mean response times and a confidence interval for a measurement period.

These values are reflected in the LQM file along with the model specification. In this way predicted mean response times for the model can be compared with measured response times for that same interval.

### 5.3 Unmanaged work and other types of interactions

We note that there is always likely to be unmanaged work on a system as well. Though we do not presently characterize this work in the model, we will eventually treat unmanaged processes as client processes that do not import interfaces. *Proxy observers* can periodically forward information about their resource consumption to the performance data server. Information about the processes can be used to create several entities to reflect their combined behaviour. For example, CPU-intensive, disk-intensive, and mixed-work entities can be introduced for each node.

The instrumentation package described in Section 3 associates an interaction type with each operation of an interface. For exported interfaces, these include synchronous, asynchronous, and forwarding. Each imported operation also has an interaction type, which includes synchronous, asynchronous, and deferred synchronous. Currently we assume that each operation of each interface is accessed and supported using a synchronous RPC. Model building extensions for the other interactions have been studied [10] and their integration into the model-building system is considered as future work.

## 6 Case study

In this section, we consider a three-tiered sample DCE application with clients and two levels of servers. All the programs have instrumentation to collect measurement data for our model building system. We have run the application for several different combinations of customer populations and server-threading levels. Layered queueing models are built and then analysed using the (MOL). The measured and predicted end-to-end response times are compared for validation. Several "what-if" scenarios for the system are explored using the analytic technique.

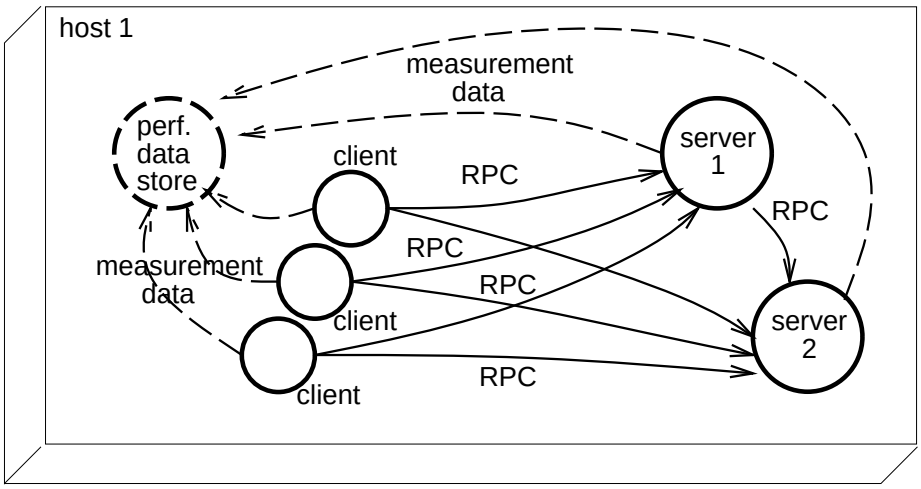


Figure 1: A sample DCE application and the performance data server

Figure 1 illustrates the sample application. Client processes (three instances are shown in the figure) invoke the operations provided by two server processes, `Server 1` and `Server 2`. `Server 1` also invokes `Server 2` as a consequence of clients' invocations of `Server 1`. All our processes in this experiment run on a single node named `host 1`. Clients alternate between invoking operations of servers, visiting the processor of their node, and "thinking". Servers handle service requests by means of the node's processor and disk. The instrumentation thread in each application process periodically forwards data to the performance data server, which is then queried by the model builder.

Table 1 describes several configurations of the distributed application system in terms of the number of client processes and the threading levels of servers. The average client think time was 16 seconds. Measured mean client response times $\bar{R}$ are given along with estimates obtained from the MOL. The reporting interval for the observer threads was one minute. Each experiment was run for one hour.

Data collected by the measurement system is used as parameters in the LQMs. The only parameter that was not measured was the disk service time, and that was estimated as 8 milliseconds. The table shows that the model-building infrastructure captures the resource demand values needed for model building. Predictive model estimates for multi-threaded servers are optimistic, compared to actual measurement. Further systematic work needs to

| Client Pop. | Server Threads | Measured $\overline{R}$ | MOL $\overline{R}$ |
|---|---|---|---|
| 1 | 1 | 0.95 | 0.95 |
| 1 | 3 | 0.97 | 0.96 |
| 5 | 1 | 1.20 | 1.21 |
| 5 | 3 | 1.20 | 1.10 |
| 10 | 1 | 1.73 | 1.72 |
| 10 | 3 | 1.56 | 1.37 |

Table 1: Mean client response times

be done to refine the approximations for the DCE environment.

The validated model can now be used to consider several alternative system configurations in order to answer "what-if" questions. For example, what is the mean response time for the system with 40 clients and server-threading levels of 3? By increasing the client population in the model and solving the predictive model we find an average response time of 15.5 seconds. This is considered too long. From detailed results we find that the CPU is highly utilized. If a second processor is added to the system and Server2 is assigned to that processor, the MOL estimates a response time of 2.7 seconds, which is considered acceptable. The behaviour of these alternatives are predicted in a matter of seconds using the analytic techniques. Making such changes to the real system and measuring their effectiveness can be expensive and time consuming. Analytic performance models can help prune the number of alternatvies for which actual changes and measurements must be made.

# Summary

We have described and demonstrated a model-building method that exploits: an instrumentation package to gather data from within application processes, a performance data server that acts as an intermediate store and analyser for performance information, and a model builder that reflects measured information in models. We used the method and tools to model a sample DCE application. The resulting model is then used to consider several

"what-if" scenarios.

The instrumentation can be included in class libraries or runtime environments to provide immediate model-building support for the environment's applications. The instrumentation captures interactions between operations of process interfaces. These interactions are currently not captured by operating systems.

Gathering performance data "by hand" and reflecting it in performance models is a time-consuming and error-prone process. We believe that automated-performance model building is necessary for the successful performance management of distributed applications.

# 7 Acknowledgements

This work has been supported by grants from IBM Canada Limited, the Natural Sciences and Engineering Research Council of Canada, and the Telecommunications Research Institute of Ontario.

# References

[1] J. Rolia and K.C. Sevcik, "The Method of Layers," *IEEE Transactions on Software Engineering,* Volume 21, Number 8, pages 689-700, August 1995.

[2] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", Special Issue of the Performance Evaluation Journal, Volume 24, Number 1-2, Pages 117-135.

[3] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software." *IEEE Transactions on Computers,* Volume 44, Number 1, pages 20-34, January 1995.

[4] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models,* Prentice-Hall, 1984.

[5] Tadeao Murata, Petri Nets, Properties, Analysis and Applications. *Proceedings of teh IEEE,* Volue 77, Number 4, pages 541-580, April 1989.

[6] R. Friedrich, J. Martinka, T. Sienknecht, S. Saunders, *Integration of Performance Measurement and Modeling for Open Distributed Processing,* Proceedings of International Conference on Open Distributed Processing (ICODP'95), Brisbane Australia, February 1995, pages 341-352.

[7] R. Friedrich and J. Rolia, "Applying Performance Engineering to a Distributed Application Monitoring System," to appear in the proceedings of the *International Conference on Distributed Processing,* ICDP '96, Dresden, Germany, Feburary, 1996.

[8] J. Rolia and V. Vetland, *Parameter Estimation for Models of Distributed Application Systems,* IBM Canada/NRC Proceedings of CASCON 1995, November 1995.

[9] D.C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communication Systems," *IEEE Journal on Selected Areas in Communication,* Volume 11, pages 489-506, May 1993.

[10] C.E. Hrischuk, J. Rolia, C.M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", Proceedings of International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95), p. 399 - 409.