Parameter Estimation for Performance Models of Distributed Application Systems

Jerome Rolia and Vidar Vetland

Carleton University
Dept. Systems and Computer Engineering
Ottawa, Ontario, Canada K1S 5B6
Email: {jar,vidar}@sce.carleton.ca

Abstract

The performance engineering of distributed applications[1] requires models that capture contention for both hardware and software resources. Layered queueing models have been proposed for modeling distributed applications but they require model parameters that can be difficult to measure directly. In particular, measures of resource demand must be found for each of the services provided by application processes.

We consider two approaches for collecting the resource demands of services. The first is a measurement-based approach where resource consumption is specifically allocated to each service. The second makes use of performance measures available from operating systems and distributed application performance monitors but employs statistical techniques to estimate the resource demands of each of the services. This paper discusses the trade-off between these two approaches and presents results from the statistical analysis.

[1] The IBM contact for this paper is Pat Finnigan, Centre for Advanced Studies, IBM Canada Ltd., Dept. 583/844, 844 Don Mills Road, North York, Ontario M3C 1V7.

1 Introduction

Distributed applications consist of many software processes distributed over a network, cooperating to accomplish some overall application goals. They offer advantages in price-performance, availability, and resource sharing. To offer a high-quality service to customers, many of these applications must satisfy performance requirements. A performance management environment [3, 7] is needed to help verify that quality of service requirements for applications are being satisfied and to provide measurements that can be used to develop predictive models for systems. The models can then be used to support performance tuning and capacity planning exercises for distributed systems. We start by giving a historical perspective on monitoring that helps to explain why measurements we require are not already available.

Performance management requires the support of a monitoring infrastructure. Mainframe environments have benefited from comprehensive monitoring systems for decades. There are three main purposes for the monitors. The first is to collect information that can be used to decide whether a host's physical resources, including processors, disks, and memory, are be-

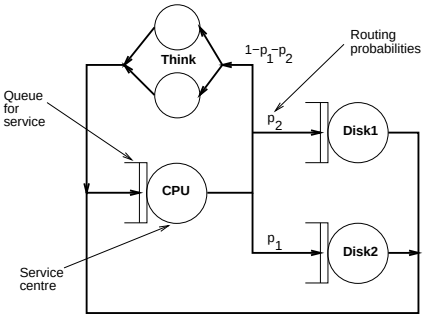

Figure 1: A Queueing Network Model (QNM)

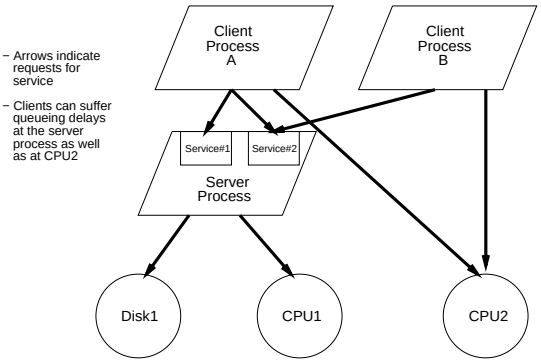

Figure 2: A Layered Queueing Model (LQM)

ing properly utilized. This is needed to lower the expense of operating a mainframe. The second purpose is to help decide whether user requests are incurring too much queueing delay or too much variation in queueing delay while competing for physical resources. If they are, the user perceives a degradation in quality of service. The third purpose is to collect the information needed to build a predictive model for the host.

Model building for mainframe environments requires workload characterization. The service demands of processes at physical resources are measured, and classes of processes with similar service time demands are identified. The demands are used as parameters in a Queueing Network Model (QNM). QNMs have been used to characterize mainframe environments for nearly two decades. An example of a QNM is given in Figure 1. Customers flow from queue to queue in the network and receive service from the service centers. Performance evaluation techniques [2, 4, 5, 9, 10] take as input a QNM and estimate mean queueing delays of the processes at the physical resources. The mean response time of a process is the sum of its average demand at the resources and the queueing delays incurred while gaining access to the resources. The model can be used to consider the performance impact of adding new work to the host, of balancing loads across resources, or of increasing the speed of a resource.

Monitoring and workload characterization are different in distributed application environments. Software processes can act both as clients and servers to other processes, and the processes of an application can span many nodes in the distributed environment. Layered queueing models (LQM) have been proposed as performance models for distributed application systems [6]. An LQM includes the parameters of QNMs but is extended to include visits between processes. Serving processes must be characterized in terms of the services they provide. This is much more information than is collected by traditional host monitoring systems. An example of an LQM is shown in Figure 2. Requests for service flow between the services of software servers and devices. A client can suffer a queueing delay when visiting a software server or a device.

Performance evaluation techniques for LQMs take into account contention for software servers and devices [6, 11, 13]. In addition to questions addressed using QNMs, these models can be used to consider the impact of moving a serving process from one node to another, of replicating a serving process, or changing its level of multi-threading.

Processes that provide service are virtual resources that must be monitored by distributed application monitoring systems. Software servers often provide many services, each with different resource demands and visits to other software servers. Distinguishing these services is important for performance modeling. Ideally, a distributed application monitor should measure the resources used by each service separately. For several reasons, this is very difficult to accomplish. In this paper we consider two approaches for estimating the resource demands and visits to other servers made by each service provided by an applica-

tion process.

The next section describes an application from the Open Software Foundation's Distributed Computing Environment (DCE) [14] and gives a corresponding LQM. The capture of the parameters needed for LQMs is the main motivation for this paper. Section 3 describes monitoring techniques for capture of resource demands of applications. Unfortunately, not all the information that is needed for predictive models is available. Sections 4 and 5 describe measurement-based and statistically based techniques for estimating the resource demands of application services. The tradeoffs between the techniques and concluding remarks are given in Section 6.

2 A DCE Application

The distributed computing environment [14] provides services that support distributed applications in heterogeneous environments. In this section we describe several DCE services, give an example application to show how the problem we discuss relates to application software, and illustrate the LQM of the application. The DCE services that are of most interest are the cell directory service, remote procedure call service, and the threading service.

The *cell directory service* (CDS) is a name server that keeps track of DCE processes. Processes that provide services to other processes prepare to do so by exporting an interface description to the CDS. These processes are classified as serving processes in an LQM. Afterwards, the serving process waits for requests for service. A process that requires a server with a specific interface queries the CDS for the addresses of servers that provide the interface. The process then chooses from the servers given by the CDS.

Process interfaces are described using an *interface description language* (IDL). The description includes a definition of the data structures used in the interface and the services provided. The services are specified as subprogram prototypes that include the name of the subprogram, the names and types of its parameters, and whether the parameters are input, output or input/output parameters. A process can export an interface with many services. Each of the services is associated with a separate subprogram. The interface description for a server that provides two services, `add_customer` and `read_customer`, is shown in Figure 3. Subprogram templates that are part of the application program are shown in Figure 4. From the psuedo-code we expect these two subprograms will have significantly different service demands.

```
interface customer_server
{
  type customer_record ....
  type customer_identifier...
  void add_customer(
    [in] customer_record in_record);
  void read_customer(
    [in] customer_identifier cust_id,
    [out] customer_record out_record);
}
```

Figure 3: Partial interface description for a server process

The remote procedure call service is used to implement communication between a calling process and a serving process. When a caller requests a service it does so by initiating a procedure call to a service provided by the interface of another process. The parameters for the call are then exchanged by the remote procedure call service. When a call is received by a serving process, it is associated with a *thread* of control that executes the corresponding subprogram of the service. Results are returned to the caller via the remote procedure call service. Serving processes have a fixed number of threads for executing these subprograms. If a thread is not available when a call arrives, the caller suffers a queueing delay. LQMs must characterize these interactions.

The DCE thread service provides an application programming interface to threads used by DCE and by application programmers. The implementation of the thread service depends in part on the operating system of the host. In some operating systems, kernel threads are available and can be used to implement the threads. Alternatively, DCE threads can be implemented within the runtime environment

```
void add_customer(
  customer_record *in_record)
{
  // Append customer record
  //   to customer file
  // Report the addition in a
  //   journal file
  // Update hash table of file
  //   locations in memory
}

void read_customer(
  customer_identifier cust_id,
  customer_record *out_record)
{
  // Find file location of customer
  //  record using the hash table
  // Read the record from the
  //  customer file
}
```

Figure 4: Subprogram pseudo-code for IDL operations

of each process.

An LQM for an application that uses the server is shown in Figure 5. Client processes are associated with dedicated workstations but compete for service from the software server. Though it is not considered in this example, software servers may also request services from other processes.

Queueing delays at a process are affected by the number of threads that are available to the process, the load on resources used by the services of the process, the service demands of the services currently being executed by other threads, and the service demands of the requested service. In Figure 4, we expect the `add_customer` service to require much more processor and input-output system resources than the `read_customer` service. The service demands of each service must be distinguished in an LQM for effective performance evaluation.

3 Monitors

Our goal is to count the visits between services in distributed applications and quantify

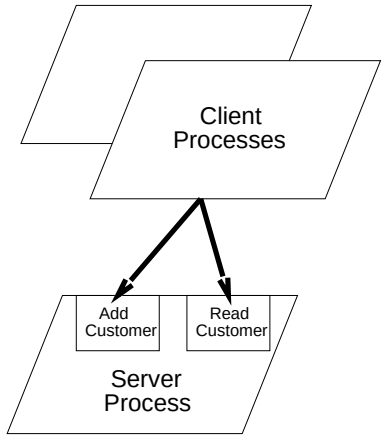

Figure 5: A layered queueing model of a DCE application

resource demands of the services. Exercisers[2] can be used to measure the resources used by process services in a controlled measurement environment. However, in this paper we focus on deployed applications. First, we consider data that are collected by operating systems and then data collected by a specific distributed application monitoring system.

Operating systems can provide us with resource consumption information such as CPU usage and physical input/output counts. Unfortunately this information is measured at the process level. Operating systems have no insight into the nature of the processing taking place within the process, so reported resource usage cannot be related directly to any single service. If thread services are implemented using operating system kernel threads, resource usage can be aggregated at the thread level. However, in DCE, threads can execute many different services. So, the provided measures are still aggregated.

We now consider a measurement system for the continuous monitoring of distributed applications. The *Distributed Measurement System* (DMS) [7] is a distributed application performance measurement system under development by the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) Special Interest Group (SIG) on Per-

[2] An exerciser is a performance tester that measures a single software component in a controlled environment.

formance [7].[3] The main purpose of DMS is to help support the performance management of DCE services and applications that make use of the services.

The data collected by DMS include statistics for metrics such as the number of times each service is used, service response times, and process resource consumption (as reported by the host operating system). Instrumented DCE runtime libraries enable each DCE process to collect running sums, moments, and histograms of performance information and periodically forward a summary to a collection process on the same node. Running sums are used to limit the amount of monitoring data captured and hence the overhead caused by DMS. Data are aggregated over pre-defined time-intervals called *process periods*. The minimum process period supported by DMS is currently 5 seconds. The collection process then periodically forwards the data over the network to a performance management node for storage and analysis. An interesting feature of DMS is that the process periods can be different for each process. This enables frequently used processes to report metrics more often than lightly used processes.

Information collected by DMS can be used to decide whether DCE services are being properly utilized. Operation response time metrics provide a measure of the quality of service for distributed applications. These two functions complement the traditional purpose of monitoring systems for mainframe environments (as discussed in Section 1). DMS can also be used to support the performance modeling of distributed applications. Unfortunately, it does not provide all the information we need for performance models.

DMS counts the number of times a process requests services of other processes. The measures are maintained on a (calling process, (serving process, service)) basis. Though it is feasible to maintain these counters on a ((calling process, service), (serving process, service)) basis, managing the resulting volume of monitor data was expected to cause too much monitoring overhead. If required, the collection of these inter-service values is easily accomplished. A more difficult problem is that most operating systems do not enable resource consumption measurements for individual services. In this paper we focus on techniques to estimate them.

In the following sections we consider techniques that can be used to estimate service resource consumption in distributed applications. The techniques make use of information reported by host operating systems and counts of service invocations available from application monitors. We assume a DMS style distributed application monitor that supports process periods.

4 Direct Measurement

The direct measurement of service resource consumption is the ideal solution for capture of performance parameter values. If high-resolution measures for CPU demand and other resource use can be collected each time a service executes, distributions for the measures can be found. The data can be maintained using running sums, moments, and histograms as in DMS. The information is useful for performance analysis. For example, a multi-modal distribution of CPU times could identify typical but different uses of a service.

We now discuss several issues that make the direct measurement of resource consumption difficult. These include the resolution of measurement counters, the overhead caused by the measurements, and the placement of instrumentation.

A prerequisite for direct measurement is operating system support for high-resolution measurements. Most UNIX environments report CPU times with a resolution in the order of 10 or 20 ms. If it takes 1 ms of CPU time to execute a service, CPU time measurement data with a resolution of 10 ms are not very helpful. To overcome poor resolution, *exerciser* techniques are often considered. An exerciser measures the resources used by many *consecutive* requests for the same service. The average amount of resources consumed by the service can then be found. This may be straightforward in a small test program or a controlled

[3] The design and implementation of DMS is under the direction of Hewlett Packard's Network Systems Architecture group.

environment, but can be difficult in real applications. Indeed, consecutive calls to a service may actually cause a program failure.

In a DCE process, direct measurement requires the instrumentation of thread services. With each thread context switch, resources used by a thread need to be allocated to the service being executed by the thread. Accounting for physical resource use is done by the operating system. Instrumentation in the thread service must be integrated with operating system resource reporting mechanisms.

Operating system calls that report on resource use introduce monitoring overhead. Measuring CPU time should not require too much overhead, but the frequent use of operating system services such as *getrusage* for measuring physical input-output counts introduces a significant overhead. The overhead caused by extensive instrumentation may prohibit measurement collection in production environments. DMS minimizes this problem by supporting different levels of instrumentation. The level can be controlled on a per process basis as an application executes.

An example of a less intrusive approach to measuring the cost of services is one that waits until a thread begins a service in an otherwise idle process. The thread can measure the current resources used by the process. If the thread is able to complete before the process is interrupted by a new call, the thread can measure the elapsed resource usage and attribute it to the one service. However, there may be some services that execute rarely, or rarely execute in an otherwise idle process. In these situations, this technique would be difficult to apply.

5 A Statistical Approach

A statistical approach for estimating service resource demands should be considered if instrumentation is not possible. Bard [1] used regression to estimate the CPU time incurred by MVS operating system supervisory calls. We will consider the use of regression to estimate the CPU time used by each application service.

As input to the regression we take a sequence of process period data for a process. The response variable r_j for a process period j could be the total CPU time, number of physical input or output operations, or visits to the services of other servers. The control variables $i_{j,k}$ are the counts of service invocations during each process period. The values form the set of linear equations shown in Figure 6. In the figure, $i_{j,k}$ is the number of times service k began service during period j. The solution of the system of linear equations gives estimates for the resource demands $(c_1 \ldots c_n)$.

$$
\begin{aligned}
c_1 \cdot i_{1,1} + c_2 \cdot i_{1,2} + \ldots + c_n \cdot i_{1,n} &= r_1 \\
c_1 \cdot i_{2,1} + c_2 \cdot i_{2,2} + \ldots + c_n \cdot i_{2,n} &= r_2 \\
&\vdots \\
c_1 \cdot i_{m,1} + c_2 \cdot i_{m,2} + \ldots + c_n \cdot i_{m,n} &= r_m
\end{aligned}
$$

Figure 6: The linear equations to be solved

The resource demands $(c_1 \ldots c_n)$ that we want to deduce may not be deterministic. They may be dependent on the data processed by the service and/or the state of the process upon service invocation. Furthermore, some services will start in one period, but end in another [12]. In these *end-effect* cases the arrival is attributed to only one of the periods, but the demand is accounted for partly in both. This causes serious problems if the process periods are very short with respect to service times. These variations will contribute to error in the predicted resource demand values.

There are several trade-offs that must be considered when preparing for the regression. Long process periods tend to even out the variation amongst control and predictor variables. However, variation is essential for a successful regression analysis. On the other hand, short process periods cause significant end-effect errors.

An advantage of the regression technique is that the input data are currently reported by operating systems and distributed application measurement systems. Furthermore, the results of the regression can be used to estimate the total server utilization and validated against its measured value. So we can always check to see if a specific application of the regression yields good results. We now consider

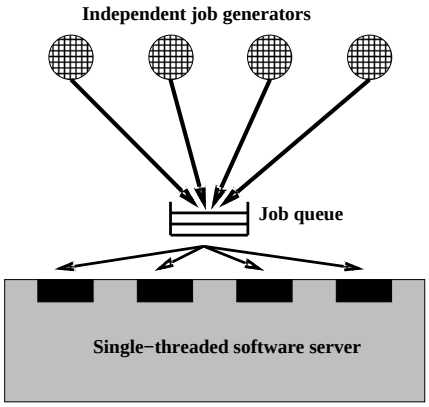

Figure 7: The structure of the simulation model

the factors that affect the accuracy of the regression and circumstances expected to give accurate results.

5.1 Experiments and Results

A discrete-event simulator was implemented to conduct experiments that cover a large parameter space. The simulator models a single-threaded software server with a certain number of service entries. For each entry there is an invocation generator. Figure 7 shows the structure of our simulation model.

Currently the model allows factorial experimentation with up to nine different factors (described later). Since the number of services to be provided by the simulated server can vary, we express parameters for services using distribution patterns. One such pattern for mean service time is `geometric(2)`, which indicates that the first service has the specified mean, the next service should use twice that value as its mean and so on. Service time distributions are based on the mean service time. For example, `Normal(0.05)` indicates that Normal variates are generated with a standard deviation of $\pm 5\%$ of the mean service time[4].

We considered three types of parameter in our experiments: system description parameters, system load parameters, and control parameters. The system description parameters were the number of service entries, the mean service time for each service entry, and the distribution of service time for each service entry. System load is described by an arrival rate and the inter-arrival time distribution for each entry. The experiment control parameters were the process period duration, the number of process periods used in the regression, and the number of replications of each experiment.

We conducted experiments to find cases where regression provides satisfactory results, then varied the experiments until unsatisfactory results were found. We avoided degenerate cases where regression is expected to fail [8], for example when arrivals between two service entries are highly correlated.

The experiments were conducted as follows:

1. Define application characteristics to simulate.
2. Vary the control parameters for each "application." These are
   - the utilization of the software server (arrival rates were derived based on service means)
   - the process period duration
   - and the number of process periods used in the regression analysis

   Two system description parameters were also considered:
   - the number of service entries
   - and the mean CPU service times for the serving entries

A metric for error is required to evaluate the accuracy of regression analysis for the different factor levels. We prefer that the services responsible for most of the server utilization be estimated most accurately. So, we decided to weight the differences between estimated and real service time means by the simulated service entry utilization, i.e., the product of number of invocations and the real mean service time. This error measure can also be interpreted as the error in our regression's estimate of the true utilization of the server.

$$E_{rel} = \frac{\sum_{\forall e} \frac{|s_{est,e} - s_{real,e}|}{s_{real,e}} \cdot i_e \cdot s_{real,e}}{\sum_{\forall e} i_e \cdot s_{real,e}}$$

[4] Normal variates with value less than or equal to zero are discarded and redrawn.

The absolute errors corresponding to different settings of the five factors were used in a five-way analysis of variance (ANOVA). The ANOVA helps us identify the factors and factor interactions that affect error most. We now present partial results from a series of experiments that were used to gain insight into the problem. Detailed results are presented for an experiment that includes the most important factors.

To illustrate the effects of different factors we describe a *basic case* with characteristics that are then changed systematically. Consider a software server that offers five services with constant (deterministic) service times. The means are 3.000, 5.400, 9.720, 17.496 and 31.493, respectively (note the constant growth factor). The "growth factor" for arrivals rate is 0.6 and the actual arrival rates are scaled so that the average utilization of the software server is 75%. The resulting inter-arrival times are 23.466, 39.111, 65.184, 108.641, 181.068. The inter-arrival time distributions are all exponential. The simulation is run for 50 periods of 10000 time units each.

A five-way analysis of variance was conducted based on this example system. Separate analyses were performed for constant and exponential service time distributions. All combinations of parameter settings were replicated three times. Mean service times were set to 1 and 3 (and then "grown" across the services of the server). The arrival rates were adjusted so that the utilization of the software server had as values 0.25, 0.5, and 0.75. The process period was varied from 10000 to 20000 time units. The number of process periods used in the regression were 50, 100, and 150. Software servers with one, three, and five services were simulated.

Both of these analyses showed that the number of services offered by the software server was by far the factor that contributed most to the variance in estimation errors. On the other hand, the utilization of the software server seemed to have no impact on the estimation errors. We found that estimation errors due to entry service time variation can be reduced by increasing the duration and/or number of process periods.

Based on the previous analyses, we designed an experiment that includes the factors of most interest. In this experiment we varied the service time distribution Normal(0.1) and Normal(0.3), the service time pattern geometric(1.8) and geometric(1.2), the inter-arrival distribution exponential and Erlang(2), the arrival pattern geometric(0.8) and geometric(1.2), and finally the number of services offered 3 and 5. By varying the interarrival time pattern above and below the value 1.0, we were able to control whether services with large service times contributed most (or least) to the utilization of the server. Table 1 shows that the service time distribution explains the biggest percentage of variation of error.

Source	SumSqr	Percent
E(y^2)	0.4637	
n E(y)^2	0.2899	
Tot Var	0.1738	100
(Service time distribution)	0.0698	40.1
(Service time pattern)	0.0024	1.4
(Inter-arrival time distribution)	0.0109	6.3
(Inter-arrival time pattern)	0.0006	0.3
(Number of operations)	0.0185	10.7
Other (sum of small interactions)		17.1
Error	0.0418	24.1
Total	0.1738	

Table 1: Analysis of variance for a five factor two level experiment

In Table 2 we show the results of 100 replications for different combinations of service time distributions and number of services. The mean error is reported along with the 90-percentile error value. For the constant service time distribution case with five services the average error of the regression with respect to simulation was 2%. Ninety percent of the errors were less than 3%. The best case is the constant (deterministic) service time distribution and the worst case is the exponential distribution. We can also see that the estimation errors increase with increasing number of services. From Table 2, if our server has five service entries and entry service times have a standard deviation within $\pm 25\%$ of the mean, our average error for estimating server utilization is only 7.3%. Ninety percent of the errors are less than 11.3%. When entry service time standard deviations are within $\pm 15\%$ of mean

values the error in our utilization estimate is an average of 5.0%.

The results are encouraging if the resources used by the services are not too data dependent. For example, a sorting routine has strong data dependencies. If it is called with many different numbers of items to sort the regression technique will probably yield poor results. The example presented in Figure 4 would be expected to provide accurate resource demand estimates.

6 Concluding Remarks

In this paper we considered two approaches for estimating the resource demands of application services. The first is a measurement-based approach where resource consumption is measured directly for each service. The second makes use of performance measures for a process as a whole but employs statistical techniques to estimate the resource demands of its services.

Direct measurement is infeasible if the resolution of the measurements are close to the actual data values being measured. With current processor speeds, a CPU timer that reports with a resolution in the order of 10-15 μs should be sufficient. As processors become faster, the resolution must improve. The DCE thread services need to be instrumented if the direct measurement of services is implemented. This is likely to be a difficult task in heterogeneous environments.

Ideally, instrumentation in thread services and operating systems will advance to the point where resource consumption can be associated with services. Until that time, statistical techniques are needed.

The statistical approach fails when there is not enough variance in the number of arrivals at the service entries, or if entry service times have high variation. We can deal with the inter-arrival time problem by decreasing the process period. However, if the period is too short, end-effect errors will increase. The process period should be at least 10 times larger than the largest entry service time to ensure an end-effect error of less than 10%. The service time distribution problem can be diminished by increasing the number of process periods that contribute to the regression.

The worst case occurs when the execution of one or more services of a software server is highly data-dependent and only low-resolution measurement tools are available. In most other cases either direct measurement and/or regression will produce the resource demands of the services with reasonable accuracy. The technique appears to work best when there are less than 10 services contributing to the utilization of the server.

Future work includes choosing appropriate subsets of process period data to improve the regression and the development of heuristics for choosing an appropriate combination of process period duration and number of process periods. The impact of context switching, measurement overheads, and nested services will also be explored.

Acknowledgements

The authors thank the anonymous reviewers for their helpful comments. This work has been supported by the IBM Canada and NSERC MANDAS project.

About the Authors

Jerome Rolia is an Assistant Professor in the Department of Systems and Computer Engineering at Carleton University in Ottawa. Jerome is a principal investigator in the IBM Canada and NSERC MANDAS project. He can be reached by email at jar@sce.carleton.ca.

Vidar Vetland is a post-doctoral researcher working on the MANDAS project in the Department of Systems and Computer Engineering at Carleton University in Ottawa. He can be reached by email at vidar@sce.carleton.ca.

References

[1] Y. Bard and M Shatzoff, "Statistical Methods in Computer Performance Analysis," in Chandy and Yeh (eds.) *Current Trends in Programming Methodology,* Vol. III, Prentice-Hall, Englewood Cliffs, N.J.

	3 services		5 services		10 services	
Service time distribution	avg.err.	90% perc.	avg.err.	90% perc.	avg.err.	90% perc.
constant	0.0073	0.0116	0.0203	0.0302	0.2920	0.4303
Normal(0.05)	0.0103	0.0182	0.0254	0.0366	0.2996	0.4513
Normal(0.15)	0.0272	0.0449	0.0502	0.0784	0.3037	0.4317
Normal(0.25)	0.0416	0.0709	0.0734	0.1129	0.3485	0.4863
Normal(0.35)	0.0575	0.0974	0.1026	0.1521	0.4256	0.5914
exponential	0.1791	0.2940	0.2936	0.4499	0.7071	0.9760

Table 2: Accuracy for cases derived from the base case

[2] Y. Bard. Some Extensions to Multiclass Queueing Network Analysis. In M. Arato, A. Butrimenko, and E. Gelenbe (eds), *Performance of Computer Systems.* North-Holland, 1979.

[3] M.A. Bauer, P.J. Finnigan, J.W. Hong, J.A. Rolia, T.J. Teorey, G.A. Winters, "Reference architecture for distributed systems management," *IBM Systems Journal,* Volume 33, Number 3, 1994, pages 426-444.

[4] J.P. Buzen. Computation Algorithms for Closed Queueing Networks with Exponential Servers. *CACM,* Vol. 16, September 1973, 527-531.

[5] K.M Chandy, U. Herzog, and L. Woo. Approximate Analysis of General Queueing Networks. *IBM J. Res. and Develop., 19,* 1 (Jan. 1975), 43-49.

[6] 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," *SCE Technical Report SCE-94-14,* Carleton University, Ottawa, Canada, June 1994. *To appear in a special issue of the Performance Evaluation Journal.*

[7] 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.

[8] R. Jain, *The Art of Computer Systems Performance Analysis,* John Wiley & Sons, 1991.

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

[10] M. Reiser, "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control," *IEEE Transactions On Communications,* August 1979, 1201-1209.

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

[12] J. Rolia and B. Lin, *Consistency Issues in Distributed Application Performance Metrics.* Proceedings of CASCON'94, Toronto, Ontario, October 31–November 3, 1994, pages 121–129.

[13] C.M. Woodside. Throughput Calculation For Basic Stochastic Rendezvous Networks. *Performance Evaluation,* Volume 9, 1989.

[14] *Introduction to OSF DCE,* Prentice Hall 1992.

Trademarks