Case Study: How Modeling Revealed Serious Performance Problems in Distributed (DCE) Systems

A. M. Khandker
masud@citi.umich.edu

T. J. Teorey
teorey@eecs.umich.edu

1 Introduction

Open Software Foundation's Distributed Computing Environment (OSF/DCE) [8] is a platform for distributed computing. DCE is a collection of tools and services for the development, use, and maintenance of transparent distributed application systems. The communication paradigm supported by DCE is the synchronous Remote Procedure Call (RPC) [1].

RPCs can be implemented on any transport layer protocol, such as TCP or UDP. RPCs over UDP can be optimized more than those over TCP. Therefore, RPCs over UDP are, in general, faster and hence of our interest in this paper.

Fundamental to the overall performance of DCE is the RPC *round trip time*, also known as *latency* or *response time*. Round trip time is the time elapsed between when an RPC is invoked and when it is returned. In this paper, we focus on the round trip time of DCE RPC.

Our earlier work describes analytic performance modeling techniques for distributed application systems [4]. Unfortunately, the techniques couldn't be validated because the model-predicted and measured round trip times didn't match. When the model predicted a decrease in the RPC round trip time, the measured round trip time showed an increase. The prediction of the model followed intuition but the actual measurement was counter-intuitive. We concluded that a performance bug in the system was causing the round trip time anomaly and investigated the reason. The result of the investigation is described this paper.

The objective of this paper is to illustrate how modeling a distributed system can reveal serious performance problems and lead to performance improvement.

We start with a background of DCE RPC in Section 2. We develop a queueing network model for RPC in Section 3 and suggest a simple extension to the Mean Value Analysis (MVA) algorithm [6] to account for parallelism present in inter-machine RPCs. We rediscover the anomaly by comparing the model-predicted round trip times with the measured round trip times in Section 4. Section 4.3 describes the anomaly. We discuss the cause behind the anomaly in Section 4.4 and suggest the fix in 4.5. Section 5 describes our conclusions and future work.

2 Background on DCE RPC

In a typical DCE configuration, potential servers export descriptions of the service they provide into the cell directory service (CDS) via the name service interface (NSI). Before making an RPC, a client obtains a description of services (e.g., by importing from the CDS) and chooses a compatible server. This process is known as the *binding process.* The end product of the binding process is a *binding handle*, which is a reference to binding information stored in the *RPC runtime.*[1] The client uses

[1] RPC runtime is a layer of software on top of the transport protocol that provides general support for RPC operations.

the binding handle for making future calls to the same remote interface.

Before issuing an RPC, a client also needs a *call handle*. A call handle keeps all the information that is related to a call in progress. The information stored in the call handle gets updated as the call progresses. For example, call handles keep track of the maximum size of the packet that can be safely sent to the server at any time. A call handle must be created before the first RPC is issued and remains in use as long as an RPC is in progress. A multithreaded client, issuing more than one RPC in parallel, needs one call handle per thread. When an RPC is finished, the call handle is cached in the binding handle. Subsequent RPCs, made with the same binding handle, *i.e.*, made to the same server interface, can reuse the cached call handle.

In RPCs over UDP, the flow is controlled by the RPC runtime. Jacobson's method of congestion control [2], which maintains windows for transmission and uses *slow-start* to open up a window, is used for flow control. The initial and the current window sizes are stored in the call handle.

When a call handle is created, the maximum UDP packet size is set to the default size of 1 Kbyte. If, for example, the maximum packet size has been set to 4 Kbytes, the runtime will discover this using the slow-start and update the call handle. Once the maximum packet size is obtained and stored in the call handle, the subsequent RPCs that reuse the call handle will not go through the slow-start. In a typical computing environment, like ours, where the maximum UDP packet size is 4 Kbytes, the round trip time of an RPC that goes through the slow-start is greater than the round trip time of an RPC that does not. Therefore, reusing the call handle can significantly reduce the round trip time.

In case of inter-machine RPCs, which are typical in distributed environments, various steps of an RPC can progress in parallel [3]. For example, the client, after handing an RPC request packet(s) to the network, may need to do some housekeeping work for the RPC before it can start servicing other requests. While the client CPU is busy doing the housekeeping for the RPC, the same RPC is progressing through the network keeping the network busy as well. Thus, we frequently encounter parallel processing in inter-machine RPCs.

Creating a binding handle has a one-time overhead time cost incurred before and during the first call to a remote interface. Subsequent RPCs to the same interface by the same client do not involve that overhead. We ignored that overhead in modeling RPCs.

RPCs that go through the slow-start take longer to finish than those that do not. In terms of modeling, the two cases may require a significantly different service demand for devices in the queueing network model. However, in a well-behaved RPC, one can expect that the slow-start will be encountered only at the beginning (and maybe once in a while during congestion) but would otherwise be rare. Therefore, we based the service demands for our model upon RPCs without the slow-start.

The service demands for the steps of an RPC performed in parallel on two devices contribute to the queueing delays on both devices, but only the device with the longer delay contributes to the round trip time. We needed to account for the effect of the parallel processing in analyzing the model.

3 The Queueing Network Model of DCE RPC

We modeled a distributed application where a DCE client application, running on one machine, communicates with a DCE server application, running on a different machine, using RPC over the connectionless user datagram protocol (UDP). The two machines are connected with a 10 Mbps Ethernet. The client is multithreaded. Each client thread issues an RPC, sleeps for a while (including zero time) after the RPC completes, and then issues another RPC. The server, also multithreaded, has a single listener thread and one executor thread per RPC in progress. The actual service procedure for an RPC does no work (*i.e.*, simply returns.)

Note that the above model does not depict a real computing environment. In reality, the client thread may do some application processing between issuing RPCs (as opposed to sleep-

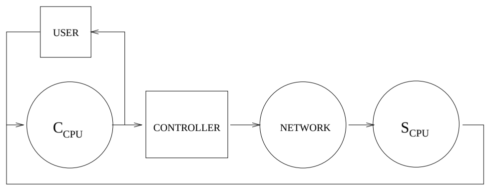

Figure 1. The Queueing Network Model for DCE RPC

ing), which demands CPU time. Also, at the server end, the service procedure usually involves real work, which demands server CPU time and adds to the queueing delay. However, the focus of this paper is the performance of RPC only. Modeling distributed systems with real workload, including the client and the server overhead, is our future goal.

The techniques in our earlier work [4] allowed us to decompose the system into logically separate components based on the natural boundaries between protocol layers, model each component separately, and finally combine the models to characterize the system as a whole. The techniques made it possible to reuse existing models for system components but required sophisticated integration methods. In this paper, we chose not to break the system in components. Instead, we built a much simpler queueing network model.

Figure 1 shows the queueing network model for the DCE RPC. The client and the server CPUs are single-server FCFS service centers with constant service times. The controllers at the client and the server machines are represented by a single delay server (shown in a single box) whose service time is the sum of all controller delays. For simplicity, the network is also modeled as a FCFS server with constant service times. The time between RPCs is assumed to be exponentially distributed.

To account for parallel processing, as mentioned in Section 2, we needed to extend the Mean Value Analysis (MVA) technique [6]. The extension to the MVA is similar to what Rolia used to model Rendezvous servers [7] and is described in the following section.

3.1 The MVA and its extension

Mean Value Analysis (MVA) is a popular technique for the analysis of closed queueing network models. It takes the following set of inputs:

- K The number of servers.
- C The number of closed classes.
- N_c The population of closed class c. (c = 1,2, ..., C)
- Z_c The average think time of closed class c. (c = 1,2, ..., C)
- $D_{c,k}$ The average service demand for class c customer at server k.

$D_{c,k}$ is the total service required by a class c customer at a server considering all visits made by the customer to that server. By definition:

$D_{c,k} = V_{c,k} * S_{c,k}$

where,

- $V_{c,k}$ The average number of visits of class c customer to server k per invocation.
- $S_{c,k}$ The average service time for class c customer when visiting server k.

The following performance measures are determined:

- $R_{c,k}$ The average residence time for class c customers at server k.
  It includes both queueing and service time.
- R_c The average residence time for class c customers.
  It is defined as $\sum_{k=1}^{K} R_{c,k}$
- X_c The total throughput of class c customers.
  It is defined as $\frac{N_c}{R_c+Z_c}$
- $Q_{c,k}$ The average queue length for class c customers at server k.
- Q_k The average number of customers at server k.
  It is defined as $\sum_{c=1}^{C} Q_{c,k}$
- $U_{c,k}$ The utilization of server k by class c customers.
- U_k The total utilization of server k.
  It is defined as $\sum_{c=1}^{C} U_{c,k}$

MVA algorithm is based on the following equations:

$$R_{c,k}(\vec{n}) = D_{c,k}(1 + Q_k(\vec{n} - 1_c)) \qquad (1)$$

$$X_c(\vec{n}) = \frac{n_c}{Z_c + \sum_{k=1}^{K} R_{c,k}(\vec{n})} \qquad (2)$$

$$Q_k(\vec{n}) = \sum_{c=1}^{C} X_c * R_{c,k}(\vec{n}) \qquad (3)$$

where,

$Q_k(\vec{n})$ is the mean total queue length at server k if the network population vector is $(\vec{n}) = (n_1, n_2, \cdot, n_C)$. $(\vec{n} - 1_c)$ denotes the population vector $(\vec{n})$ with one class c customer removed.

Given the performance of N-1 users, equations 1 through 3 are sufficient to compute the performance of N users. Because the performance with no user, i.e., $(\vec{n}) = (\vec{0})$ can be easily computed, performance of any number of users can be computed iteratively. This method of solving the model is known as the exact mean value analysis (as opposed to the approximate mean value analysis described later.)

To account for the parallel processing in RPCs the service demand at the devices can be thought to be composed of two phases. In the first phase, the job remains in a device, *i.e.*, cannot proceed to another device until the phase one service is completed. After phase one, the job is released and can move to other devices for service. But the device the job has just left must still complete the phase two service before it can serve other jobs.

Phase two service demands contribute to the queueing delays and are included when queueing delays are calculated. Thus, the service demand for our model is the summation of phase one and two service demands. i.e.,

$D_{c,k} = D'_{c,k} + D''_{c,k}$

where,

$D'_{c,k}$ Phase one service demand for class c customer at server k.

$D''_{c,k}$ Phase two service demand for class c customer at server k.

Because phase two service demands do not contribute to the RPC round trip time, we subtract them from the response time calculation. So, we modify the response time of a class to be

$R_c = \sum_{k=1}^{K} R_{c,k} - D''_{c,k}$

4 Model Validation

We consider three primary factors affecting RPC performance. First, the amount of data sent with the RPC, which represents various kinds of RPCs that might be present in the real world. Second, the number of client threads, which represents the level of concurrency in a system. Third, the time between RPCs, which plays a significant role in the workload. We design experiments using various values for these factors and measure the round trip time for each of these cases. The values for the factors in our experiments are as follows: We consider NULL RPCs and RPCs that generate 1, 2, 3, 6, and 12 packets of request data and only zero byte reply data. (RPCs with non-zero byte reply data can be modeled easily by including the additional service demands for such reply packet(s).) We chose the number of client threads to be 1, 3, 6, and 9. The time between RPCs is assumed to be an exponentially distributed random variable with mean varied from 0 to 216 ms depending on the round trip time of the RPC.

The same values for the factors were given as input to the model. The service demand for the model is obtained by measuring the time required to complete different steps of an RPC. Table 1 shows the service demands at the service centers for different kinds of RPCs obtained by measuring the system. The details of the measurement, *i.e.*, how to define and measure RPC steps, can be found in our earlier work on performance of DCE RPC [3].

4.1 Comparing Model Predictions with Actual Measurements

Round trip times were measured for each individual thread by letting the threads complete a total of 100 RPCs. The experiment was repeated 15 times and the average round trip time with the minimum variance was used. With a few exceptions, the minimum variance was observed with the minimum average.

Table 2 compares the model-predicted round trip times (column 4) with the measured round trip times (column 5) at different levels of the factors we consider. Column 6 shows the %

RPC type	Service Demand (millisecond)			
	C_{CPU}	*CONTROLLER*	*NETWORK*	S_{CPU}
NULL	2.98 (0.82)	1.17	0.20	3.41(1.21)
1 packet	3.54 (0.82)	1.53	1.31	3.64 (1.21)
2 packet	4.20 (1.10)	1.53	2.54	3.84 (1.21)
3 packet	4.99 (1.79)	1.53	3.58	4.84 (1.94)
6 packet	9.51 (3.61)	3.06	7.15	10.46 (5.26)
12 packet	18.18 (7.98)	4.59	14.29 (3.57)	18.63 (10.23)

Table 1. Service demand of various types of RPC The numbers shown are the combined service demands for phase one and two. Phase two service demands are shown in parenthesis.

Error calculated as:

$$\% \text{ Error} = \frac{(\text{Measured value} - \text{Model predicted value}) \times 100}{\text{Measured value}}$$

High % Errors reveals inconsistency between the model and the system.

The modified system (column 7) is discussed in section 4.5.

4.2 Simulation

Having failed to validate the analytic model, we developed a simulation model for RPC round trip time using GPSS/H [9]. The round trip times prediced by the simulation matched the predictions from the analytic model, within 12% in all cases.

The simulation considered the steps of RPC separately, as opposed to the notion of a single service demand at each device in the analytic model. The nature of simulation automatically accounted for the parallel processing in the system for calculating round trip times. In short, the simulation model captured the dynamics of RPCs more accurately than had been possible with the analytic model.

Nevertheless, we couldn't validate the analytic model using the simulation model because both models were based on the same assumption of how RPCs work and used the inputs obtained from the same set of experiments. However, the close match between the analytic and the simulation model helped us conclude that the analytic model was accurately modeling our understanding of how RPCs work and the analysis had no significant flaw.

This led us to believe that the actual system was at fault. We believed that there was an anomaly in the system.

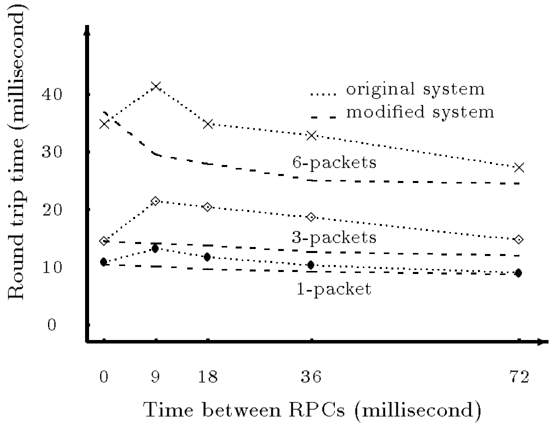

Figure 2. The round trip time anomaly. Measured round trip time of RPCs with three client threads in the original system are shown by the dotted lines. The round trip time goes up when delays are introduced and then gradually goes down. For comparison, the anomaly free behavior from the modified system is shown by the dashed lines.

4.3 The Anomaly

To understand the anomaly, let us consider a single row in Table 2. The average measured round trip time of back-to-back (*i.e.*, with zero inter-RPC time) RPCs with 2-packet request data with six client threads was 23.80 ms. The model predicted 22.76 ms – very close to the measured value. With the introduction of 3 ms delays between RPCs, the model predicted a small decrease in the round trip time (20.55 ms), while the measurement showed a substantial increase in the average (33.90 ms) as well as in the standard deviation (18.60). Intuitively, the delay between RPCs should have lessened the load on the system and decreased the round

RPC type	# of client threads	Think time(ms)	Round trip time (microsecond) Model predicted	Measured original avg	Measured original (std)	% Error	Measured modified avg	Measured modified (std)	% Error
1-pack	1	0	7.99	7.90	0.20	-1.14	7.90	0.20	-1.14
		3	7.99	8.00	0.30	0.13	8.10	0.20	1.36
	3	0	11.23	10.80	1.80	-4.02	10.40	0.70	-8.02
		3	10.46	14.60	7.20	28.35	10.10	0.90	-3.57
		9	9.64	13.20	5.40	27.01	9.60	1.00	-0.37
		18	9.07	11.70	4.20	22.45	9.20	1.10	1.38
		36	8.63	10.30	3.60	16.23	8.70	0.90	0.83
		72	8.34	9.00	2.20	7.35	8.40	0.70	0.73
	6	0	19.78	18.50	1.10	-6.90	18.70	0.80	-5.76
		3	17.41	23.90	10.80	27.18	15.60	2.50	-11.57
		9	14.16	20.70	8.10	31.58	13.40	2.40	-5.70
		18	11.75	17.90	7.70	34.36	11.80	2.80	0.43
	9	0	30.31	28.20	1.70	-7.49	29.50	2.40	-2.76
		3	27.37	34.20	16.30	19.98	26.00	2.60	-5.26
		9	22.03	31.00	17.00	28.95	21.00	3.40	-4.89
		18	16.56	25.50	10.40	35.07	16.50	3.40	-0.35
3-pack	1	0	11.21	11.20	0.30	-0.09	11.10	0.30	-0.99
		6	11.21	11.10	0.30	-0.99	11.10	0.30	-0.99
	3	0	15.93	14.50	2.10	-9.84	14.40	1.10	-10.60
		9	14.10	21.40	10.50	34.09	14.10	1.50	-0.03
		18	13.27	20.40	9.80	34.95	13.70	1.80	3.14
		36	12.50	18.60	9.90	32.77	12.60	1.60	0.76
		72	11.95	14.70	7.10	18.72	12.00	1.30	0.44
	6	0	27.28	25.00	1.60	-9.13	25.70	1.80	-6.15
		9	21.61	42.20	20.00	48.79	19.50	3.10	-10.82
		18	18.36	37.20	20.10	50.63	17.50	3.90	-4.94
		36	15.35	35.70	24.40	56.99	15.40	4.10	0.30
		72	13.36	22.30	12.00	40.07	12.70	1.70	-5.23
	9	0	41.10	52.70	16.30	22.01	42.90	11.60	4.20
		18	26.90	72.90	37.40	63.10	26.20	5.60	-2.67
		36	19.93	53.10	31.80	62.47	22.20	8.50	10.24
		72	15.31	39.10	26.50	60.84	16.90	6.90	9.39
6-pack	1	0	21.31	21.60	2.60	1.34	21.30	1.90	-0.05
		6	21.31	21.30	1.60	-0.05	21.50	1.60	0.88
	3	0	30.91	34.90	10.50	11.44	36.90	11.70	16.24
		9	28.66	41.40	15.40	30.78	29.50	6.20	2.85
		18	27.22	34.90	8.70	22.00	27.90	5.20	2.44
		36	25.52	32.90	9.10	22.42	25.00	3.50	-2.09
		72	23.96	27.30	7.50	12.23	24.50	3.80	2.20
	6	0	54.13	57.90	11.60	6.50	51.40	10.70	-5.32
		9	47.67	58.20	14.00	18.10	49.30	10.80	3.31
		18	42.64	55.30	15.00	22.90	45.10	10.70	5.47
		36	36.01	53.10	20.70	32.19	36.40	7.80	1.08
		72	29.83	41.80	17.50	28.64	30.40	8.00	1.87
	9	0	82.56	60.70	7.50	-36.02	81.20	15.20	-1.68
		18	66.41	63.80	13.90	-4.10	67.70	16.10	1.90
		36	53.68	54.30	22.60	1.14	57.00	14.80	5.82
		72	39.32	53.20	33.40	26.08	39.40	12.40	0.19

Table 2. Model Validation. Error in model-predicted round trip times with respect to the original and the modified system.

trip time. Therefore, the actual behavior of the system is anomalous. The round trip times of back to back RPCs in a multithreaded DCE client are smaller than the round trip times of RPCs when delays are introduced between the RPCs. The anomaly is not present for NULL RPCs or RPCs with 12-packets of data. We wanted to investigate this also.

Figure 2 depicts this anomalous behavior graphically for 1-packet, 3-packet, and 6-packet RPCs with three client threads. The round trip time in the original system increases as some delay is introduced between RPCs and then gradually decreases as shown by the dotted lines. For comparison, the dashed lines show the round trip times in the modified system discussed in section 4.5.

After considerable investigation over two months, we found the reason behind this anomaly.

4.4 The Reason

Initially, we thought that the thread package, particularly the scheduler could be the cause of the anomaly. To precisely locate the actual source of delay, *i.e.*, whether the server, the client, or both contributed to the elevated round trip time, we installed the Berkeley Packet Filter (BPF) [5] in a machine on the same Ethernet to watch network packets. We discovered that in the case of non back-to-back RPCs, the number of small packets, packets that one would see at the beginning of a slow-start, is much greater than the number of similar packets for back-to-back RPCs. We realized that the random invocation of the slow-start was the problem, but still didn't know why slow-start was being invoked at random.

After browsing through the code, we found that the current RPC runtime implementation allows only a single call handle to be cached in a binding handle. This is not a problem for single-threaded clients because there could only be one RPC in progress; no more than one call handle is ever created for a binding handle. On the other hand, a multithreaded client issuing more than one RPC in parallel creates more than one call handle for reasons that were described in Section 2. Of these call handles, only one can be cached at a time.

The availability of only one cache for a call handle is not a problem for back-to-back RPCs, even in a multithreaded client. In the case of back-to-back RPCs, each thread puts the call handle into the cache and immediately grabs it for the next RPC. Therefore, call handles are never kept in the cache long enough to cause contention for the cache among threads. Each thread always finds a call handle in the cache to use; the one found is the one that has just been put back into the cache.

When delays are introduced between RPCs, the call handles are kept in the cache for a random period of time. If other threads complete their RPCs during that period and find the cache to be full, they destroy the call handles. Destroying call handles requires that new ones be created when needed in the future. RPCs, issued with new call handles go through the slow-start and take longer to finish. This explains why the round trip times of back-to-back RPCs are smaller than the round trip times of the RPCs when delays are introduced between two RPCs.

The reason for the anomaly not being observed by NULL RPCs is that the NULL RPC generates only one packet; therefore, the slow-start does not come into play. In fact, the anomalous behavior should not be present for RPCs that result in a single UDP packet. In RPCs with very large data, *e.g.*, 12-packet RPCs, the anomaly is present but not visible. In such cases, the reduction in the round trip time due to the introduction of delay between RPCs subsided the penalty for the slow-start.

4.5 The Fix

A possible solution for eliminating the anomaly is to allow more than one call handle to be cached with each binding handle. This can be implemented with a linked list. Another solution is to keep the information about the packet size for the RPC in the binding handle instead of the call handle. We maintain that the latter approach is more elegant. The information about the packet size truly depends on the machines involved in the communication (not the call that is made) and thus the information belongs in the binding handle. With this approach, all threads will be forced to slow down if one thread detects congestion.

The solutions we suggested above need substantial modification in the RPC runtime source code and a total recompilation. Because our access was limited to a pre-release version of the source code, we fixed the problem differently. We saved all packet size related information before destroying a call handle and restored it back to a newly created one. We reran the experiments with the modified system. Some sample results are tabulated in column 7 of Table 2. The round trip times for the modified system are very close to those obtained from the model. The absolute maximum error is 17.50%. The average and the standard deviation of the percent error are 0.18 and 6.31.

5 Conclusion and Future Work

We developed a reasonably accurate performance model for DCE RPC over UDP. The DCE system we modeled had a performance bug. The mechanism to improve the performance of RPC by caching call handles did not work for a multithreaded client. Thus, the RPC round trip times in the system were larger than what they should have been had the system been properly optimized.

Usually, when a model fails to match measurements in the real system, we look for errors in the model. However, the close match between the analytic and the simulation model, that we built subsequently, suggested that the bug was in the system and not in the model. The notion that the system was at fault seemed very reasonable due to the presence of anomalous behavior in the system.

Had the anomaly not been present, *i.e.*, had the round trip times been consistently greater than the model-predicted times in all cases, we might have rejected the model. We have learned that this may not always be the case. A system that deviates too much from its model warrants investigation, which can often help build an improved system.

The bug was discovered in the process of building the model, which emphasized the importance of building performance models throughout the life cycle of these systems.

Our future goal is to include the client and the server in the model and build a model for DCE applications. A methodology has already been suggested in our earlier work [4].

References

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] Van Jacobson. Congestion Avoidance and Control. *Proceedings, ACM SIGCOMM'88 Stanford, CA*, pages 314–329, August 1988.

[3] A. Masud Khandker, Peter Honeyman, and Toby J. Teorey. Performance of DCE RPC. In *Proceedings, 2nd International Conference on Services in Distributed and Networked Environments, Whistler, British Columbia*, pages 2–10, July 1995.

[4] A. Masud Khandker, Jerome A. Rolia, and Toby J. Teorey. Performance Modeling of the Distributed Computing Environment. In *CD ROM proceedings, the fifth annual CASCON conference, Toronto, Canada*, November 1995.

[5] S. McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. *USENIX*, pages 259–269, January 1993.

[6] Martin Reiser and S. S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[7] J. A. Rolia. Predicting the Performance of Software Systems. Technical report, CSRI Technical Report 260, University of Toronto, Canada, 1992.

[8] Ward Rosenberry, David Kenny, and Gerry Fisher. *Understanding DCE.* O'Reilly and Associates, Inc., 1992.

[9] Thomas J. Schriber. *An Introduction to Simulation Using GPSS/H.* John Wiley & Sons, 1991.