Performance Modeling of the Distributed Computing Environment

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

J. A. Rolia
jar@sce.carleton.ca

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

1 Introduction

Distributed application systems are often implemented using services provided by middleware environments. Examples of midware environments include the Open Software Foundation's Distributed Computing Environment and the Object Management Group's Common Object Request Broker Architecture. In these environments, application processes request services both from devices (such as processors, disks, and networking elements) and from software servers. Communication in these environments is complex, and performance is difficult to predict.

At the Center for Information Technology Integration (CITI), at the University of Michigan, we run the Open Software Foundation's Distributed Computing Environment (OSF/DCE) [10], 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 DCE architecture (Figure 1) is layered bottom-up from the most basic, or suppliers of services, to the highest-level consumers of services, the applications. DCE provides runtime support for distributed applications. This runtime support hides much of the underlying complexities of distributed environments. This support is basically a layer of software that provides standard interfaces to the upper layer applications, helping develop portable applications. The communication paradigm supported by DCE is the synchronous Remote Procedure Call (RPC) [2]. RPC relies on the transport layer services provided by the operating system. The transport layer, in turn, uses the network layer services below. These layers of software provide boundaries for decomposing the system into smaller components, enabling us to build less complex models of those individual components. The problem, then, becomes how to integrate these individual models to obtain the overall performance of the system. In this paper, we focus our attention on the DCE RPC, but the methodology we describe is applicable to any distributed system with similar architecture, *e.g.*, Sun's ONC.

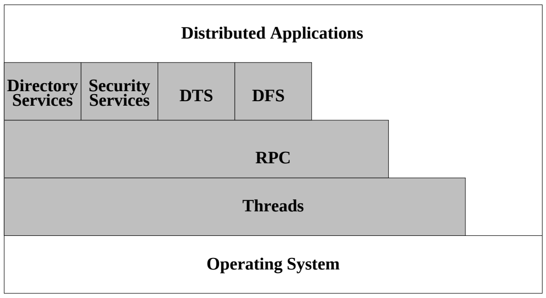

Figure 1. DCE layered architecture. Each layer relies on the services provided by the lower layers, the suppliers of services, and provides standard interfaces to the upper layers, the consumers of services. The shaded regions show the components of DCE.

Figure 2 shows the decomposition of a DCE

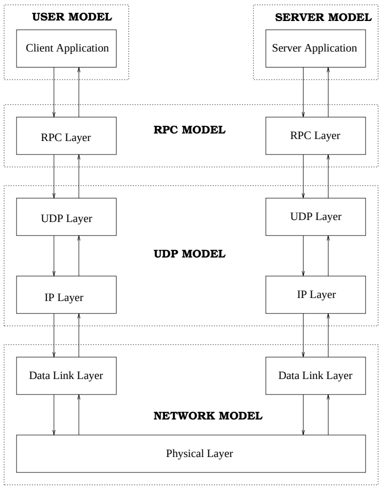

Figure 2. Protocol layering. Shows decomposition of client/server communications along the communications protocol boundaries. Models include one or more protocol layers.

application along the protocol layer boundaries and five different models that include one or more protocol layers. Queueing delays may arise at any of these layers when the layer is implemented as a software server. In such a case, that layer is to be reflected as a server in our predictive performance model.

For simplicity, we don't include the highest level of DCE services, *i.e.*, the directory, security, time, and file service, in the model. DCE RPC can run both on TCP and UDP; we consider RPCs that run on UDP only. The UDP and IP layers are merged together (*i.e.*, the service demand for the IP layer is included in the UDP layer service demand) and a single model called the UDP model is built. The bottom two layers are modeled with NetMod [1], a network modeling tool. For the rest of the paper, we call the model for the lowest two layers the network model. (This is not to be confused with the ISO network layer.)

Fundamental to the overall performance of DCE is the *round trip time*, also known as *latency* or *response time*, of its RPC mechanism – the time elapsed between when an RPC is invoked and when a response is returned. The round trip time includes the overhead associated with the RPC layer as well as delays at the layers below (Figure 2). In this paper, we focus on the round trip time of DCE RPC.

Because our primary focus for this paper is not the DCE client or server applications, we don't model them. We use synthetic applications so simple that their presence can simply be ignored during model validations. The RPC and UDP layers in a single machine contend for the same CPU and thereby add to the queueing delay. We develop a Layered Queueing Model (LQM) [8] for the RPC and UDP layers that takes care of such situations. The Method of Layers (MOL) [8] is used to analyze the LQM. The network model is analyzed by a network modeling tool called NetMod. The performance of UDP and RPC depends on the delay at the network. So the network layer delay must be an input to the LQM. Again, the delay at the network is dependent on the packet throughput. Packet throughput depends on the UDP and RPC layer delay and must be given as an input to NetMod. For that reason, we need to solve these two models iteratively until the round trip time converges.

The following sections describe related work, how DCE RPC works, the modeling and analysis technique used (layered queueing models, method of layers, and NetMod), and the model integration. The remaining sections discuss validation of the model and future work.

2 Related Work

Rolia et al. studied the performance behavior of an application server executing within DCE on an AS/400 communicating with other processes using RPC [9]. They developed a layered queueing model that reflects the contention for the AS/400's processor and contention for the TCP/IP, listener, and executor processes on the AS/400. A significant amount of measured information, including the number of resends and TPC/IP time per packet, were used to create the analytic model. Their model allowed them to vary those values and to estimate the impact on performance of DCE implementation alternatives. Our model is geared to characterizing RPC round trip times in general environments.

3 Background on DCE RPC

In DCE, a potential server exports a description of the service it provides into the cell directory service (CDS), via the name service interface (NSI). Before issuing an RPC, a client obtains descriptive information (e.g., by importing from the CDS) and chooses a server from a set of compatible ones. This process is known as *binding.* At this point, the client is said to be *partially bound* because it knows only the network address (*e.g.*, IP address) of the server machine but does not know the transport layer address (*e.g.*, TCP/UDP port number) of the server within that machine. The transport layer address is obtained by communicating with the server's *endpoint mapper* at the beginning of the first RPC. Once this is done, the binding information is complete and the client is said to be *fully bound.*

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

Creating a binding handle has a one 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 ignore that overhead in our performance model.

RPCs can be implemented on any transport layer protocol, such as TCP or UDP. We confine our model to RPCs over UDP only. When an RPC is made over the UDP layer, flow is controlled by the *RPC runtime.* We assume that Jacobson's method of congestion control [4], which maintains windows for transmission, is used for flow control. The round trip times in our model assume no retransmission.

Any amount of data can be sent with an RPC. This data is referred to as the *request data.* Similarly, any amount of data can be received back from the call. This is known as the *reply data.*

DCE supports user level threads for achieving better concurrency. When threads are employed, a single client process can issue one RPC per thread. As a result, in a multi-threaded process more than one RPC can be in progress at any instant.

When a client application calls a procedure in a remote interface, control is transferred to the stub module for that interface in the caller's address space. The client machine then performs the following steps:

C_1 : Creates a *call handle* from the binding handle and obtains the negotiated transfer syntax.

C_2 : Marshals the arguments using the transfer syntax from step 1.

C_3 : Copies the request data into the call packet(s). If the data size is greater than the current window size, copies a portion of it into the call packet(s).

C_4 : Hands call packets to the UDP layer for transmission. Note that one call packet may generate more than one network packet.

C_5 : Handles interrupts generated by the Ethernet controller for transmitting each network packet. If more data needs to be sent, marshals the next window of data and copies into the call packet(s). Sets the wakeup timers (in case retransmission is warranted) and waits for the acknowledgement or reply.

C_6 : As the packets containing the acknowledgement or the reply comes in, handles interrupts for the arrival of each network packet. Lets the IP and UDP layers extract the RPC packet and resumes the client process waiting for the packet.

C_7 : Copies the received data into the caller's address space. If the data marks the end of the RPC, returns the call to the caller; otherwise, waits for more data. If an acknowledgement is received, and more data needs to be sent, goes to step 4 to send more data.

A DCE server performs the following steps:

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

S_1 : Handles the interrupts generated by the Ethernet controller upon arrival of each network packet.

S_2 : Extracts the RPC packet and transfers control to the listener thread of the server process waiting for the RPC.

S_3 : The listener thread wakes up one of the executor threads from a pool of threads waiting to provide service. The listener thread, then, waits for more incoming packets. The executor thread, after some initial setups, unmarshals the arguments.

S_4 : The executor thread calls the actual RPC (i.e. the service procedure).

S_5 : Sends the result packet(s).

S_6 : Handles interrupts generated by the Ethernet controller for transmitting reply packets. The listener thread checks for more incoming packets. If nothing can be done at this point, sets some wakeup timers, blocks itself, and waits for the next RPC.

Note that both the client step C_5 and the server step S_6 are performed in parallel with other devices servicing the RPC. Also, the interrupt handlings are asynchronous and are overlapped with the network transmission time. In a typical distributed environment, these steps do not (directly) contribute to the round trip time of the RPC. However, processing these steps consumes resources, and adds to the queueing delay. The model must account for these overlapping services.

4 Modeling and Analysis Techniques

In this section, we introduce the modeling and analysis techniques that we use later to model DCE. We discuss the Layered Queueing Model (LQM), a technique for modeling hierarchical software systems, the Method of Layers (MOL), a method for analyzing an LQM, and NetMod, a tool for network performance modeling.

4.1 The Layered Queueing Model

In hierarchical software systems that contain one or more layers of software servers, a process can act as both customer and server, so it is possible for one process to visit another. While one process is acting as a server to another process, it is possible that they are competing for a single hardware device, e.g. the CPU. This type of situation can be modeled by the Layered Queueing Model (LQM) described by Rolia [8]. In an LQM, processes having statistically identical behavior form a group or class. Groups use devices but can also request services from and provide services to other groups.

4.1.1 Model Parameters

The input parameters for an LQM are a superset of those required for closed separable queueing network models. They are:

G, K		Set of groups and devices
L		The number of software levels in the LQM hierarchy
G_n	$\forall n \epsilon 1 \cdot \cdot L$	The set of groups at level n of hierarchy
N_g	$\forall g \epsilon G$	Population of group g
$V_{g,h}$	$\forall g, h \epsilon G$	The average number of visits from group g to group h
$V_{g,k}$	$\forall g \epsilon G \forall k \epsilon K$	The average number of visits from group g to device k
$S_{g,k}$	$\forall g \epsilon G \forall k \epsilon K$	The average service time of a visit from group g to device k
Z_g	$\forall g \epsilon G$	The think time of group g
ψ_j	$\forall j \epsilon G \cup K$	The queueing discipline of group or device j

For serving groups, the think time input parameter, Z_g, is assumed to be zero.

4.2 The Method of Layers

The Method of Layers (MOL), also proposed by Rolia [8], provides performance estimates for LQMs. The MOL divides an LQM into two types of models. The *software contention model* describes the software relationships in the process architecture. The *device contention model* describes each group's device usage. The two models are solved alternately, with the solution of one helping to determine some of the parameters of the other.

The MOL requires groups of processes in a software contention model to be associated with levels in a hierarchy. A process that requests service from another is one level up in the hierarchy. Requests for service that span more than one level can be mapped onto LQM, but require some special techniques. It is also necessary that requests be acyclic, i.e. there may be no cycles in the graph. The MOL decomposes the software contention model into several two-level models and treats each of them as a closed Queueing Network Model (QNM). The device contention model can have only two layers; all groups belonging to the top level and all devices to the bottom one. The MOL treats this configuration as a closed QNM as well.

As mentioned earlier, the MOL alternates between the software contention model and the device contention model for solution. The initial values are obtained assuming no device and software contention. Then the software model is considered – beginning with the top two layers, treating them as a closed QNM. A variation of approximate Mean Value Analysis (MVA) called *linearizer* [3] is used to solve the QNM. Values obtained from the top two layers are used in the solution of the second two layers and so on. Several iterations are required among all two-layer models within the software contention model before the response times of the processes at the top layer converge to a fixed value. Then the device contention model is considered using the results from the software contention model as inputs. Once again the linearizer is used to solve the model. Results are then fed back to the software contention model. Several iterations are also required between these two models before the final results can be obtained.

Different servers may have different scheduling disciplines and service time distributions. Linearizer uses different *residence time* expressions for different kinds of servers. This residence time expression is valid only for servers that satisfy certain assumptions of scheduling discipline and service time distribution. Among the allowable combinations of scheduling disciplines and service time distributions for hardware devices, DELAY and FCFS are of interest. A service center with DELAY scheduling discipline behaves like one with an infinite number of servers, so the customers never need to wait for service. A first come first served (FCFS) service center with only deterministic service time distribution is used in this paper. The residence time for such a server is given by Reiser [7].

Unlike hardware devices, software processes may interact with each other in complex ways. Rolia discusses the residence time expressions for FCFS, Rendezvous, Multiple-Entry, Multi-Server, SYNC, and DELAY software servers [8]. Among them the Rendezvous and DELAY servers are of interest. The Rendezvous server is a FCFS process with two phases of service. A caller is released after the first phase, but the server cannot begin to provide service again until it completes the second phase.

In addition to the metrics provided by MVA, MOL computes the average response time of a group.

4.3 NetMod

NetMod is an easy to use capacity planning tool that helps engineers and managers planning and designing large-scale complex computer networks. NetMod users can quickly evaluate numerous complex LAN/WAN configurations.

The inputs to NetMod are the topology parameter, the workload characteristics, and the characteristics for the device/transmission media. NetMod uses a graphical interface with icons representing networks and their components. A NetMod user starts with a blank window and specifies a network topology by clicking the mouse on icons, selecting components from icon menus, and drawing arrows between the components. NetMod supports popular LAN technologies – token ring and Ethernet and some of their variations – as well as special network components such as routers, bridges, gateways, and adapter cards. The workload parameter is specified by the packet arrival rate and packet size. NetMod has a built-in user workload model that can suggest these parameters, given the user profile. NetMod provides the default values for the device/transmission media characteristics, which the user can change easily. NetMod's output includes component utilization, throughput, and packet delays which it calculates from mathematical models that use closed-form analytical techniques.

To use the full capability of NetMod together with MOL, some integration is required. Ideally we would like to have a single graphical interface to describe both the LQM and the Network topology so that NetMod models can be invoked with the right topology during the analysis. This integration is part of our future work. In this paper, we modeled a simple Ethernet environment and just used the Ethernet model from NetMod. NetMod uses Lam's equation [6] to find the Ethernet delay.

5 The DCE Model

We model 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 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 one. The server, also multithreaded, has a single listener thread, and one executor thread per RPC in progress. The actual service procedure for an RPC (server step S_4 in section 3), called by the executor thread, 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 making RPCs (as opposed to sleeping), which demands CPU time. Also, at the server end, the service procedure usually involves real work – therefore demands server CPU time and adds to the queueing delay. However, the focus of this paper is the performance of DCE, which is affected by the performance of the RPC layer and all layers below. For this reason, we keep the service demand of the client and the server application minimal, so that they can be ignored during the analysis of the model.

We consider three primary factors affecting RPC performance. First, the amount of data sent with the RPC, representing 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 levels for these factors and measure the round trip time for each of these cases. The levels for the factors in our experiments are as follows. We consider RPCs with 0, 1392, 2872, 4016, 8032, and 16064 byte request data and only with 0 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).) The levels for the number of client threads are 1, 3, 6, and 9. The time between RPCs is assumed to be an exponentially distributed random variable with mean varied from 0 to 72 milliseconds depending upon the round trip time of the RPC.

5.1 RPC and UDP model

Figure 3 shows the layered queueing model for the RPC and UDP layer. Groups of processes are shown as rectangles, devices as circles and requests for services, also known as visits, as directed arcs. The RPC model in Figure 2 is represented by two process groups – RPC_c on the client machine, and RPC_s on the server machine. Similarly, UDP_c and UDP_s represents the client and server portion of the UDP model. A hypothetical communication process group, COM, accounts for the network layer delay. RPC_c and UDP_c contend for the CPU in the client machine. Similarly, RPC_s and UDP_s contend for the server CPU. The COM process requests services from the controller device $CONT$ and the network NET.

Figure 3 describes the set of groups and de-

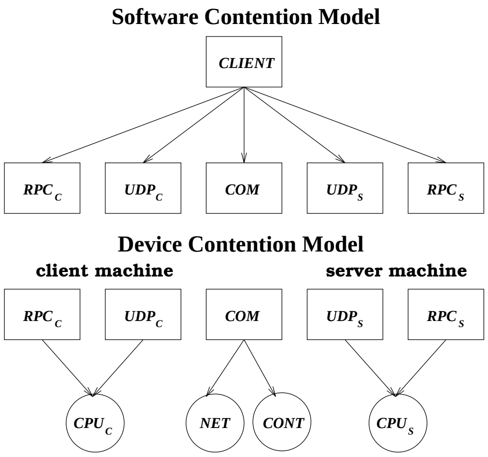

Figure 3. The layered queueing network model for DCE. It is divided into the software and device contention models. Groups of processes are shown as rectangles, devices as circles, and requests for services as directed arcs. The subscript c and s indicate whether a process or device belongs to the client or server machine.

vices, the number of software levels, and the set of groups at any level – *i.e.*, G, K, L, and G_n input parameters described in section 4.1.1. In all cases, the average number of visits (parameter $V_{g,h}$ and $V_{g,k}$) is 1, when such visits are shown by the directed arcs, otherwise the number of visits is zero.

The client and the server CPU are Rendezvous service centers with constant service times. We extend the MOL to allow Rendezvous hardware devices. (Originally, only groups were allowed to have such a service discipline.) In the extended MOL, every device can have two phases of service. The combined service time of these two phases are used to calculate the queueing delay at the device but only the service time of the first phase is included in calculating the time needed to obtain the service. The controllers at the client and the server machines are represented by a single DELAY server whose service time is the sum of all controller delays.

We also introduce a queueing discipline called MODEL. The MODEL is a DELAY server or device whose parameters are initially unknown but become known as the analysis progresses. The device NET in Figure 3 is of type MODEL.

Although we decompose DCE in layers of software, these layers do not correspond to layers of software servers. (A single client thread of execution cuts across the RPC, UDP, and IP layer and there is no software queueing delay at these layers.) The layered queueing modeling technique, which has been developed primarily for software with layers of servers, allows us to model this kind of situation by considering all these layers as DELAY servers. The population of DELAY servers is equal to the population of the process that requires service from them.

In our model, we consider all service providing groups, *i.e.*, RPC_c, UDP_c, COM, UDP_s, and RPC_s, as DELAY servers.

All other input parameters, *i.e.*, N_g, $S_{g,k}$, and Z_g are considered to be factors in our experimental design and their values are case dependent. The service demand of processes at various devices depends on the RPC type. In earlier work we measured the time it takes, on average, to perform various steps of a single inter-machine RPC [5]. In this paper, we regard the time allocated to a particular step of an RPC as the service demand for the hardware device associated to that step. The service demands of all steps pertaining to a layer are combined to obtain the service demand of that layer.

Table 1 shows the service demands of different layers at various devices for all RPC types considered in our experiment. The populations of $CLIENT$ and RPC_c are equal to the number of client threads. A DCE server spawns one thread for each RPC up to a maximum of ten threads. For our experiments, where the maximum number of client threads is nine, the population of RPC_s is therefore also equal to the number of client threads. The populations of UDP_c, UDP_s, and COM are set to 1.

5.2 Network model

The network topology is simple – two machines connected by a 10 Mbps Ethernet. The workload parameters, *i.e.*, the packet arrival rate and mean packet size are obtained from the LQM. The default values for the device/transmission media characteristics supplied by NetMod are used.

RPC type	Service Demand (millisecond)					
	$RPC_c \rightarrow CPU_c$	$UDP_c \rightarrow CPU_c$	$COM \rightarrow CONT$	$COM \rightarrow NET$	$RPC_s \rightarrow CPU_s$	$UDP_s \rightarrow CPU_s$
NULL	1.44 (0.73)	1.53 (0.26)	1.17	0.20	1.04 (0.64)	1.10 (0.28)
1 packet	1.49 (0.73)	1.70 (0.26)	1.53	1.31	1.06 (0.64)	1.29 (0.28)
2 packet	1.56 (0.73)	1.88 (0.63)	1.53	2.54	1.06 (0.64)	1.38 (0.73)
3 packet	1.60 (0.73)	2.05 (1.00)	1.53	3.58	1.06 (0.64)	1.47 (1.17)
6 packet	2.93 (1.40)	4.10 (1.99)	3.07	7.15	1.78 (1.04)	2.94 (2.34)
12 packet	4.26 (2.82)	6.85 (5.08)	4.60	10.72 (3.57)	2.51 (2.86)	5.73 (4.17)

Table 1. Service demand on hardware devices. Left of the arrows are the processes and right of the arrows are the devices. Subscript c and s represent the client and the server machines. The portion of the service demand that is not included in the RPC round trip time is shown in parentheses.

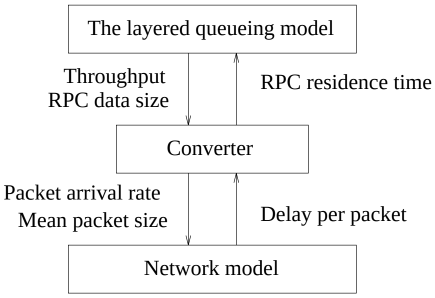

Figure 4. Integration of the layered queueing network model with NetMod. The procedure is iterative. Output from the layered queueing model is used as input to NetMod (the network model) and *vice versa*. Some conversion is required before outputs of one model can be fed into the other.

The network is assumed to be dedicated to the two machines. The experimental measurement was done at night when the outside traffic on the Ethernet was negligible.

5.3 Integration

Figure 4 shows the integration of the layered queueing model of RPC and UDP with the network model.

The performance of the LQM depends on the network delay. In particular, LQM needs an input value for the service demand for the device called NET shown in Figure 3. On the other hand, we cannot get the network layer delay unless we know the RPC throughput; otherwise we cannot specify the packet arrival rate, which is an input to the network model. So we solve these two models iteratively using the intermediate result from one model as input to the other.

First, we assume a zero network delay and solve the LQM. Then the throughput from the LQM is used as input to NetMod. However, the output parameters from the LQM gives the throughput of the RPCs. These RPC throughputs need to be converted to the packet throughput and size of the packets for NetMod. (One RPC may generate a number of network packets depending on the request or reply data sizes.) This is shown by the converter box in Figure 4. Therefore, we need the following additional information for the converter. NULL RPCs generate two 108-byte Ethernet packets, one for the request and one for the reply. RPCs with 0, 1392, 2872, 4016, 8032, and 16064 byte request data generate 1, 2, 3, 6, and 12 Ethernet request packets with a mean packet size of approximately 1400 bytes and 1, 1, 1, 2, and 4 reply packets with a mean packet size of 108 bytes.

Once the network delay is calculated using the network model, the packet delay is converted back to the RPC delay by the converter. That delay is now given as an input to the MOL for the subsequent iteration and used as the residence time at the NET. The iteration goes on until the round trip time from the LQM converges.

5.4 Validation

Ideally, we would like to validate the model described above by comparing the round trip times obtained from the model with the experimentally measured round trip times. Unfortunately, the measured round trip times of RPCs in our environment are counter-intuitive in many cases. This RPC round trip time anomaly is discussed further in section 5.5. For that reason, we couldn't compare our model predicted round trip time to the measured round trip time. So, we wrote a simulator and used the simulated round trip time to compare with the results of the analytic model. The simulated round trip times closely match the experimentally measured round trip time in those cases when the latter does not show any anomalous behavior.

Table 2 compares the model estimated round trip time with the simulated ones for different levels of the factors. The error is calculated as:

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

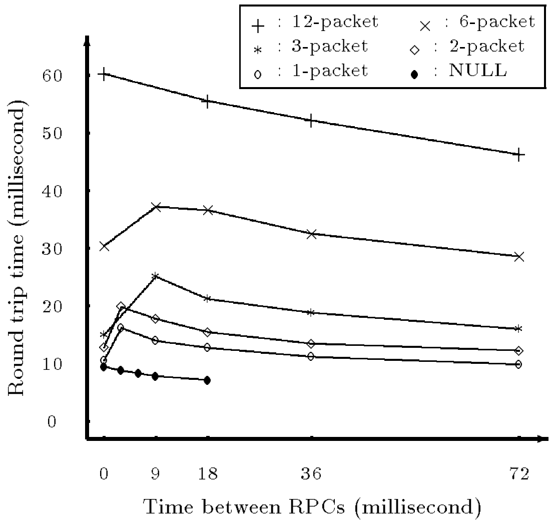

Figure 5. Measured round trip time of RPCs with 3 client threads. Intuitively, the round trip time should never go up when the delay between RPCs is increased. Actually, the time goes up as soon as some delay is introduced and then goes down gradually as the delay is increased.

5.5 Round trip time anomaly

In this section, we describe measured round trip times that defy intuition.

Figure 5 shows the round trip times obtained from actual measurements for NULL, 1-packet, 2-packet, 3-packet, 6-packet, and 12-packet RPCs with 3 client threads. For example, the average measured round trip time, of making back-to-back (*i.e.*, with zero inter-RPC time) RPCs with 3-packet request data, is 14982 microseconds. Now the introduction of delays between RPCs should, intuitively, lessen the load on the system and decrease the round trip time. But the measurement shows that the time actually goes up first and then goes down gradually as we increase the time between RPCs. This anomalous behavior is observed for 1, 2, 3, and 6 packet RPCs but not for the NULL and 12-packet RPCs shown in Figure 5. The behavior is similar with 6 or 9 client threads (see Table 2).

At this time, we don't know why such is the case, but are continuing to investigate.

6 Conclusion and Future Work

Modeling distributed systems based on client/server architecture by decomposing the system along the communication protocol layers, modeling individual components separately, and finally integrating them is a feasible option. Over 90% of the model predicted round trip times have less than 10% difference from the corresponding simulated round trip times.

In this paper, we combined features from a network performance modeling tool and a software performance modeling tool. An integration of these two to make a single tool for distributed application systems is our future work.

Although both models we considered in this paper are analytic, they could easily be a simulation or measurement model. The same integration technique will work as long as the input to and the output from the model are the same and consistent convergence criteria

RPC type	# of client threads	Think time(ms)	Round trip time (microsecond) Measured avg	(std)	Simulation predicted	Model predicted	% Error
NULL	1	0	6081	(243)	6465	8262	-27.80
		3	6152	(229)	6465	8044	-24.42
	3	0	9496	(346)	11168	11856	-6.16
		3	8827	(1473)	8829	10830	-22.66
		6	8359	(1506)	7895	10127	-28.27
		9	7866	(1455)	7523	9637	-28.10
		18	7191	(1310)	7095	8825	-24.38
	6	0	19117	(576)	22325	19005	14.87
		9	13143	(3171)	13517	13779	-1.94
		18	10125	(2698)	10103	11290	-11.75
		27	8975	(2715)	8027	10089	-25.69
	9	0	28662	(724)	33483	27786	17.01
		18	15567	(4280)	16356	15219	6.95
		27	12795	(4629)	11720	12568	-7.24
		36	10283	(3753)	10005	11148	-11.42
1-pack	1	0	8290	(152)	8363	9177	-9.73
		3	8431	(317)	8363	8946	-6.97
	3	0	10585	(432)	11896	12928	-8.68
		3	16220	(6437)	10247	11912	-16.25
		9	13995	(5120)	9375	10682	-13.94
		18	12781	(4949)	9100	9806	-7.76
		36	11234	(4114)	8774	9124	-3.99
		72	9900	(2972)	8602	8735	-1.55
	6	0	21122	(622)	23768	20347	14.39
		3	21964	(8806)	20814	18247	12.33
		9	23230	(10091)	15545	15158	2.49
		18	20256	(9190)	11328	12562	-10.89
	9	0	31680	(810)	35644	29403	17.51
		3	35272	(12377)	32635	26685	18.23
		9	33138	(13127)	26646	21907	17.79
		18	28782	(12924)	18476	16877	8.65
2-pack	1	0	9950	(246)	9989	10696	-7.08
		6	10150	(328)	9989	10353	-3.64
	3	0	12858	(570)	13843	14938	-7.91
		3	19870	(7576)	12449	13930	-11.90
		9	17808	(7085)	11667	12651	-8.43
		18	15511	(6411)	11053	11674	-5.62
		36	13518	(5098)	10518	10866	-3.31
		72	12303	(4208)	10372	10372	0.00
	6	0	25806	(887)	27627	23489	14.98
		3	26044	(9953)	24401	21365	12.44
		9	28709	(12761)	18301	18128	0.95
		18	27938	(12642)	14481	15232	-5.19
		36	22578	(10691)	11987	12721	-6.12
		72	15414	(6446)	11012	11255	-2.21
	9	0	38725	(1269)	41431	33618	18.86
		3	43783	(14768)	38387	30969	19.32
		9	42550	(16597)	33046	24110	27.04
		18	40422	(17132)	23444	20726	11.59
		36	33014	(16284)	15735	15442	1.86
3-pack	1	0	11440	(361)	11326	11981	-5.78
		6	11454	(276)	11326	11649	-2.85
	3	0	14982	(492)	15510	16800	-8.32
		9	25120	(12970)	13625	14392	-5.63
		18	21244	(11544)	12903	13297	-3.05
		36	18867	(10213)	12277	12354	-0.63
		72	16024	(8594)	11847	11755	0.78
	6	0	30013	(1233)	30924	27098	12.37
		9	42369	(23405)	22832	21174	7.26
		18	42402	(23165)	17488	17819	-1.89
		36	34519	(19956)	14376	14756	-2.64
		72	25410	(16107)	12810	12898	-0.69
	9	0	40256	(2121)	46370	40421	12.83
		18	56798	(30829)	28436	24992	12.11
		36	49893	(28609)	18794	18474	1.70
		72	36751	(24712)	14094	14407	-2.22
6-pack	1	0	21969	(386)	22041	23126	-4.92
		6	22165	(410)	22041	22771	-3.31
	3	0	30445	(952)	30659	32098	-4.69
		9	37251	(8968)	27390	29221	-6.68
		18	36636	(9100)	26269	27432	-4.43
		36	32566	(9191)	24886	25432	-2.19
		72	28635	(7333)	23515	23761	-1.05
	6	0	57479	(3519)	61201	53110	13.22
		9	64805	(14525)	52325	45682	12.70
		18	63870	(14739)	43505	40311	7.34
		36	53763	(17185)	35004	33724	3.66
		72	42196	(16067)	27336	28025	-2.52
	9	0	85974	(5438)	91771	81073	11.66
		18	92055	(20564)	73536	62153	15.48
		36	83282	(21745)	58006	47561	18.01
		72	60230	(22896)	37193	34836	6.34
12-pack	1	0	34995	(1276)	35997	40115	-11.44
		18	35137	(1383)	35997	39188	-8.86
	3	0	60297	(1473)	55438	56450	-1.83
		18	55586	(10408)	48143	50612	-5.13
		36	52203	(12115)	45787	47240	-3.17
		72	46311	(10269)	41410	43677	-5.47
		144	40964	(9052)	39478	40892	-3.58
	6	0	113127	(7481)	110768	97378	12.09
		36	93634	(22884)	77784	69906	10.13
		72	76996	(22606)	61402	57535	6.30
		144	58394	(21719)	48060	47734	0.68
	9	0	165957	(10564)	166135	150149	9.62
		36	138312	(68317)	130176	112049	13.92
		72	116405	(32535)	98763	81820	17.16
		144	96281	(65286)	62937	58817	6.55
		216	71002	(56171)	51623	50619	1.94

Table 2. Error in model predicted round trip time with respect to simulation.

are met. Precise input and output parameters and convergence criterion for component models need to be specified.

The network was included in the MOL as a hardware device. MOL needs to be extended to integrate models of process groups. The two models we considered do not contend for the same device (*e.g.*, the NetMod model does not demand any CPU time). Techniques need to be devised when two separate models contend for a single resource – a situation that will be frequent if process group models are developed separately and need to be integrated.

We only considered a single class of jobs. The model needs to be extended for multiclass jobs. The cause of the anomalous behavior of RPC round trip time will be investigated.

Acknowledgements

We are greatful to Peter Honeyman for his advice, criticism, and direction for this work. Special thanks to Mary Jane Northrop for editing.

This work was supported by a research partnership with IBM.

References

[1] David W. Bachmann, Mark E. Segal, Mandyam M. Srinivasan, and Toby J. Teorey. "NetMod: A Design Tool for Large-Scale Heterogeneous Campus Networks". *IEEE J. on Selected Areas in Communications (JSAC)*, 9(1):15–24, January 1991.

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

[3] K. Mani Chandy and Doug Neuse. "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems". *Communications of the ACM*, pages 126–134, February 1992.

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

[5] A. Masud Khandker, Peter Honeyman, and Toby J. Teorey. "Performance of DCE RPC". To appear in *Proceedings, 2nd International Conference on Services in Distributed and Networked Environments, Whistler, British Columbia*, July 1995.

[6] S. S. Lam. "A Carrier Sense Multiple Access Protocol for Local Networks". *Computer Networks*, 4(1):21–32, February 1980.

[7] Martin Reiser. "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control". *IEEE Transaction on Communications*, pages 1199–1209, August 1979.

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

[9] J. A. Rolia, M. Starkey, and G. Boersma. "Modeling RPC Performance". *Proceedings, CASCON'93*, pages 677–689, October 1993.

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