# Performance of DCE RPC

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

*P. Honeyman*
honey@citi.umich.edu

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

## 1 Introduction

The Open Software Foundation's Distributed Computing Environment (OSF/DCE) platform for distributed computing has the goal of facilitating application level interoperability in a heterogeneous environment [4]. DCE is a collection of tools and services for the development, use, and maintenance of transparent distributed systems using the client/server architecture. The communication paradigm supported by DCE is the synchronous remote procedure call (RPC). DCE also supports multithreading within an address space for concurrency. To the programmer, RPC is similar to a procedure call in a sequential program, therefore no major paradigm shift is required on the part of the programmer, unless threads are employed for better concurrency.

Like any real-world computer system, the performance of DCE is crucial to its success. In measuring performance, we focus first on the latency and throughput of RPC, which is fundamental to the overall performance of DCE. Round trip time, also known as the *latency*, or *response time*, includes the overhead associated with the RPC layer, as well as delays at the transport and network layer. We also report on the effect of using application level DCE threads for improving the throughput.

Second, we measure the average completion time of various steps of single inter-machine RPCs. The figures tell us how much time is spent in each of these steps and allow us to estimate how much faster the RPC would be if certain improvements were made, enabling us to isolate the most beneficial optimizations.

We also investigate how the completion time of various steps contribute to the the round trip time. Often, the steps at the client and the server are performed concurrently or overlap with network transmission time. Hence, not all RPC steps performed by the client, server, or the network contribute to the round trip time of the RPC. We show the steps in our environment, that do contribute to the round trip time for RPCs with various data sizes. The close match between the sum of the completion times of these steps and the measured round trip time verifies the results presented in this paper.

The round trip time that we found by summing up the completion time of steps on round trip path does not take queueing delays into account, but our results can be fed into a more elaborate model that does account for queueing delays. The measurement work described in this paper provides groundwork for a larger goal of developing more elaborate performance models.

## 2 Related Work

Schroeder's detailed measurement of the elapsed time at various steps of Firefly RPC serves as a model and inspiration for our work [6]. Other related performance work includes the performance of DCE RPC and threads on Sun SPARCstaions by Dasarathy et al. [1], and the performance of Sun RPC by Rosenblum [5]. Performance comparison of several different kinds of RPCs was done by Rabenseifner et al. [3].

## 3 How RPC Works in DCE

Before a DCE client makes a remote procedure call, it obtains a *binding handle.* A binding handle is a reference to binding information stored in the *RPC runtime*[1], which identifies the server that will process the call.

In a general DCE setup, potential servers export descriptions of the service they provide into the cell directory service (CDS), via the name service interface (NSI). A client imports such information and choses a server from a set of compatible ones. This process is known as the *binding process.* 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.* This binding information can be used by the RPC runtime for making future RPCs to the remote interface.

There are other ways of creating a binding handle. The user might supply the binding information as a string. In such cases, a binding handle can be created by parsing the string. This process is referred to as *string binding.*

Creating a binding handle has a one time cost incurred before and during the first RPC. In the case of string bindings, the client need not import information from the CDS, but the first RPC still has some extra overhead.

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.*

### 3.1 Steps in an RPC

In this section we describe the steps involved in an inter-machine DCE RPC that generates only one request packet and one reply packet. A few more steps are involved in the case of larger request data and are described in the following section. With large data, the client sends a portion of data and waits for an acknowledgement before sending more data. For brevity, we assume that there is always a single reply packet and that there is no loss of packets (*i.e.*, no retransmission). Our description excludes the binding procedure and the extra communication incurred in the first RPC. Some of the procedure names and their orderings used in the description are specific to the implementation that we are working with.

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. Figures 1 and 2 show the RPC steps followed by the client and the server. Each line segment, marked from $C_1$ through $C_{21}$ and from $S_1$ through $S_{14}$, shows a step of an RPC with a single request and a single reply packet. Step $C_{22}$, $S_{15}$ and $S_{16}$ are needed for large data. Each step consumes some CPU time. The numbers, which are the completion times of the respective steps in microseconds, are discussed in section 4.3. The steps are usually performed in the order of increasing step number, except when transmission interrupts are handled (*i.e.*, steps $C_{15}$ and $S_{13}$).

In the case of large data, there are situations when a jump to a different step occurs. These jumps are shown by arrows in Figures 1 and 2. (The length of a line segment drawn in the figure is not in scale with the time that it takes to complete the corresponding step.) The CALL and RETURN labels imply calling a C function, when the instrumentation is done outside the function

[1]RPC runtime provides general support for RPC operations and is layered on top of the transport protocol.

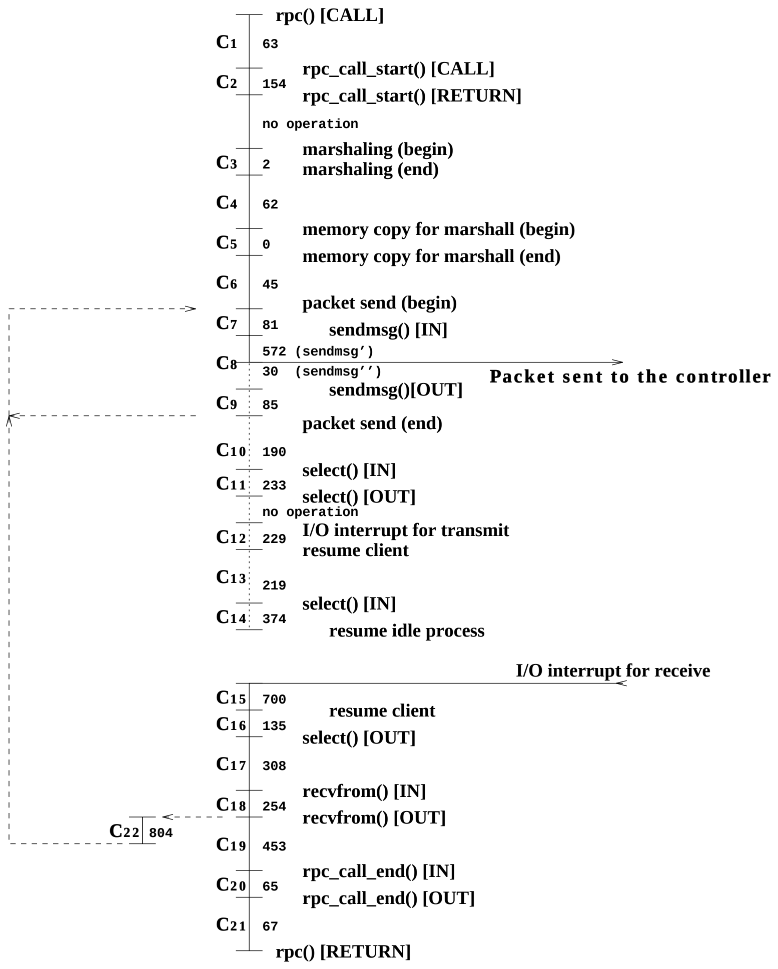

**Figure 1. Steps followed by a client issuing an RPC.** Steps shown by the dotted lines are not usually included in the round trip time.

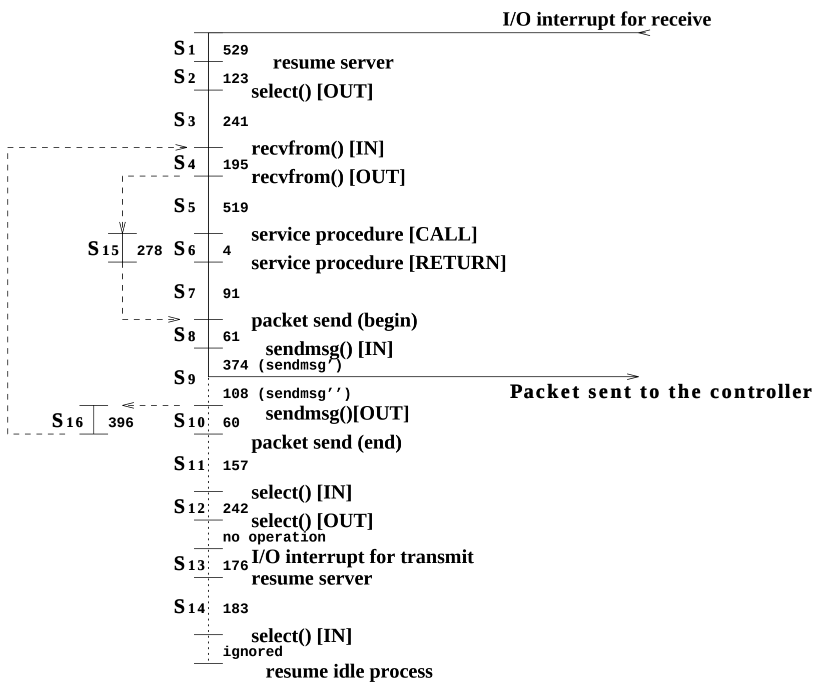

**Figure 2. Steps followed by a server servicing an RPC.** Steps shown by the dotted lines are not usually included in the round trip time.

call, *i.e.*, when the time includes the function call overhead. The IN and OUT labels imply calling a C function, when the instrumentation is done inside the function call, *i.e.*, when the time does not include the function call overhead. The begin and end labels mark the beginning and end of an activity. We now elaborate on some steps of special interest.

### 3.1.1 Client steps

$C_2$: **Start -** A *call handle* is created from the binding handle and the negotiated transfer syntax is obtained. (The current implementation supports only the Network Data Representation (NDR) as the transfer syntax. No negotiation is done.) The call handle is used to hold the state of the RPC.

$C_3$: **Marshal -** The arguments are marshalled into the call packet using the transfer syntax from step $C_2$. The outgoing data in the call packet is represented by an *iovector*, which indicates (possibly several) locations and sizes of data. No data copying is done at this point.

$C_5$: **Memcopy -** The request data is copied into the call packet.

$C_6$: **Queue -** The request packet is queued at the transmission queue.

$C_8$: **Sendmsg -** The packet is sent using a *sendmsg* system call. It is the step where a device driver routine is called for writing the packet out to the controller (*i.e.*, the packet is actually handed over to the controller for transmission). For this reason, we further divide this step into two - called Sendmsg$'$ and Sendmsg$''$; the dividing point being the moment the first device driver call within the *sendmsg* system call returns. Sendmsg$''$ is not included in the round trip time of the RPC.

$C_{11}$: **Select -** By doing a *select* system call, the client checks for any incoming packet. This is a non-blocking select call.

$C_{12}$: **SendIntH -** The interrupt, generated by the Ethernet controller for the transmission of the packet is handled. This step is performed asynchronously with other steps. For example, the interrupts may come during step $C_{11}$. In that case, because of the high priority of interrupts, step $C_{12}$ will be performed by preempting step $C_{11}$.

$C_{13}$: **SetTimers -** Nothing can be done at this point. Before making a blocking system call, some wakeup timers are set.

$C_{14}$: **Idle -** A blocking *select* is called that causes the the idle process to be dispatched and run.

$C_{15}$: **RecIntH -** Reception of the reply packet by the Ethernet controller generates an interrupt. The interrupt is handled at this step. The packet is copied into the kernel buffer from the controller. The IP and UDP layer code calculates the checksum, extracts the packet, finds the client process which had been waiting for the packet and the process is resumed.

$C_{18}$: **Recvfrom -** The packet is received by the runtime RPC by the *recvfrom* system call.

$C_{20}$: **End -** The runtime RPC marks the end of the call.

$C_{21}$: **Free -** The memory is freed and the RPC returns.

#### 3.1.2 Server steps

$S_1$: **RecIntH -** Server handles the interrupt for the request packet, similar to the client step $C_{15}$.

$S_2$: **Wakeup -** The control is transferred to the user space of the application server process. The listener thread starts to execute.

$S_4$: **Recvfrom -** The packet is received by the runtime RPC in a buffer in the user space by the *recvfrom* system call.

$S_5$: **Finding what to do -** Upon receiving the RPC packet, the listener thread in the server 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, unmarshalls the arguments and calls the service procedure.

$S_6$: **Service procedure -** The actual RPC (*i.e.*, the service procedure) is called.

$S_9$: **Sendmsg -** The result packet is sent using the *sendmsg* system call. Similar to step $C_8$, we can divide this step in Sendmsg$'$ and Sendmsg$''$.

$S_{11}$: **End execution -** The executor thread goes back to the pool.

$S_{12}$: **Select -** By doing a non-blocking *select* system call, the listener thread of the server checks for any incoming packet. This is similar to client step $C_{11}$.

$S_{13}$: **SendIntH -** This interrupt, generated by the Ethernet controller for the transmission of the reply packet, is handled. Like step $C_{12}$, this step is performed asynchronously.

$S_{14}$: **Select -** If nothing can be done at this point, some wakeup timers are set.

$S_{15}$: **Idle -** A blocking *select* is called that causes the the idle process to be dispatched and run.

### 3.2 Steps with large data

When an RPC with large data is made over the UDP layer, flow control is done in the RPC runtime layer. The flow control method could be implementation specific; The DCE implementation[2] closely follows Van Jacobson's method of congestion control [2]. The runtime RPC maintains a transmission window which defines the maximum size of unacknowledged data at any point. The size of the window at the beginning of an RPC is set to be 4 KB. The size is doubled after each acknowledgement is received, up to a maximum of 32 KB. Whenever the client has more transmission data than the current window size, it sends one window of data and requests an acknowledgement. On receiving the data, the server knows that it has received only a fraction of the data and sends an acknowledgement back, as per the client request. Sending an acknowledgement by the server does not involve steps $S_5$, $S_6$, or $S_7$. Instead, the server performs step $S_{15}$ to construct the acknowledgement packet and then goes to step $S_8$ for sending the packet. This is shown in Figure 2 by the arrow from the end of step $S_4$ to step $S_{15}$ and then to $S_8$. Upon receiving a packet in step $C_{18}$, the client goes to step $C_{19}$ only when the RPC is completed (*i.e.*, the packet received is the last of the reply packets). When the client has more data to be sent, it follows step $C_{22}$ and goes back to $C_7$ where more data gets sent.

The RPC runtime calls the *sendmsg* system call with a maximum of 4KB of data. Which means that when the window is bigger than 4KB, *sendmsg* is called more than once. This situation is shown in Figure 1 by the arrow from $C_9$ back to $C_7$.

If the size of the request data is bigger than the window size, only one window of data is copied in step $S_6$ before the first send. The remaining data is copied at a later time, when the client will otherwise be idle (*e.g.*, when the client is waiting for an acknowledgement). This can increase the completion time of the step $C_{10}$ in Figure 1 by the amount of memory copying time.

### 3.3 Network packets generated by an RPC

When an RPC with a large amount of request data is made, the RPC runtime layer takes a maximum of 4016 bytes of data at a time, adds 80 bytes of header information, and gives it to the UDP layer. So the maximum size of an RPC packet is 4096 bytes. The UDP header adds 8 header bytes to it, chops the large packet into a maximum of 1480 bytes per packet, and gives them to the IP layer. An IP header of 20 bytes makes a 1480 byte UDP packet, 1500 bytes — the maximum size of an Ethernet packet. If we look at the packets generated by an RPC with a large amount of call data, we see sequences of three packets; the first carries 1392 bytes of call data, the second one carries 1480 bytes, and the third 1144 bytes, resulting in the 4016 byte transfer. Given the size of data to be sent with an RPC, it is possible to calculate the number of IP packets generated for that RPC (assuming no retransmission). For example, if an RPC has 6000 bytes request data, the first 4016 bytes will generate three packets. The remaining 1984 bytes will require two more packets — the first having 1392 bytes and the second one having the remaining 592 bytes. Thus, the total number of packets will be five.

## 4 Measurement of RPC Performance

We measured RPC performance while making RPCs between a client and a server on two separate machines in the same DCE cell. The cell is composed of several IBM RISC System/6000 machines of model 520 and 530. All of them run AIX 3.2.4 and IBM version 1.2 of OSF DCE 1.0.2 and are connected via 10 Mbps Ethernet.

A client/server interface is defined using the Interface Definition Language (IDL) specified by OSF/DCE. The server exports one procedure for each RPC request data size we considered in our

[2]The DCE implementation, we are experimenting with, is from IBM, Austin.

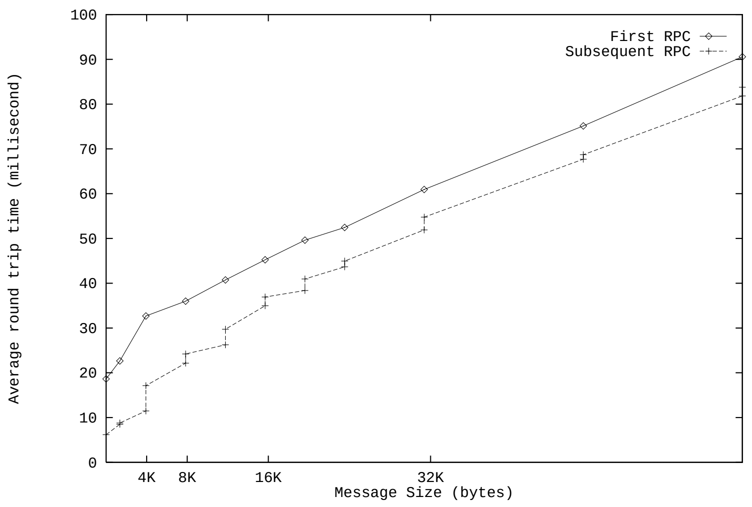

**Figure 3. RPC round trip time without any security and zero byte reply data.** The first RPC always takes longer. The round trip time of a subsequent NULL RPC is 6.17 milliseconds. RPC with 64256 byte request data (generating 48 ethernet packets) takes 81.82 milliseconds, and is able to achieve a throughput of 6.28 Mbps. Jumps in the round trip time of subsequent RPCs are shown at various data sizes when the last data byte overflows into a new network packet.

test. (The server exports several other procedures that are not relevant to this paper. Altogether, the server implements 27 remote procedures. The extra procedures have no measureable effect on RPC performance.) The procedures accept various sizes of request data as a single parameter and do not do anything (*i.e.*, simply return). All data is passed as a fixed size array of characters. A procedure called *null()*, which takes no arguments, does nothing, and produces no result, is used to measure the base latency of the RPC mechanism.

The client uses *string binding*, which eliminates the necessity of communication with the CDS and the endpoint mapper during the first RPC. The client application runs on a model 520 with 48 MB memory, while the server runs on a model 530 with 64 MB memory.

## 4.1 Latency

We measure the round trip time of an RPC with various data sizes. Figure 3 shows the average round trip time for doing the first and subsequent RPCs. The first NULL RPC takes 18.63 milliseconds to complete; subsequent ones take 6.19 milliseconds. The round trip time increases with data size, but not linearly. Sometimes, the last byte of the data may result in one more packet, *e.g.*, with 1393 byte data. (1392 bytes is the maximum RPC data that can be sent with one Ethernet packet.) That extra packet with one byte data can result in one more send — receive cycle, *e.g.*, at 4017 byte data. (4016 bytes is the maximum RPC data that can be sent with the first *sendmsg* call.) The jumps in the round trip time for subsequent RPCs are shown at 1393, 4017, 8033, 12049, 16065, 20081, 24097, 32129, 48193, and 64257 byte data size.

| number of caller threads | Calls to Null | | Calls to 4016byte send | |
|---|---|---|---|---|
| | completion time (sec) | RPC/sec | completion time (sec) | throughput (Mbps) |
| 1 | 6.262 | 160 | 11.687 | 2.749 |
| 2 | 3.348 | 299 | 6.976 | 4.605 |
| 3 | 3.202 | 312 | 6.288 | 5.109 |
| 4 | 3.238 | 309 | 5.805 | 5.534 |
| 5 | 3.244 | 308 | 5.892 | 5.453 |
| 6 | 3.251 | 308 | 5.801 | 5.538 |
| 7 | 3.270 | 306 | 5.889 | 5.456 |
| 8 | 3.283 | 305 | 5.856 | 5.486 |
| 9 | 3.298 | 303 | 5.859 | 5.483 |
| 10 | 3.301 | 303 | 5.987 | 5.366 |

**Table 1. Average completion time and throughput for 1000 RPCs.**

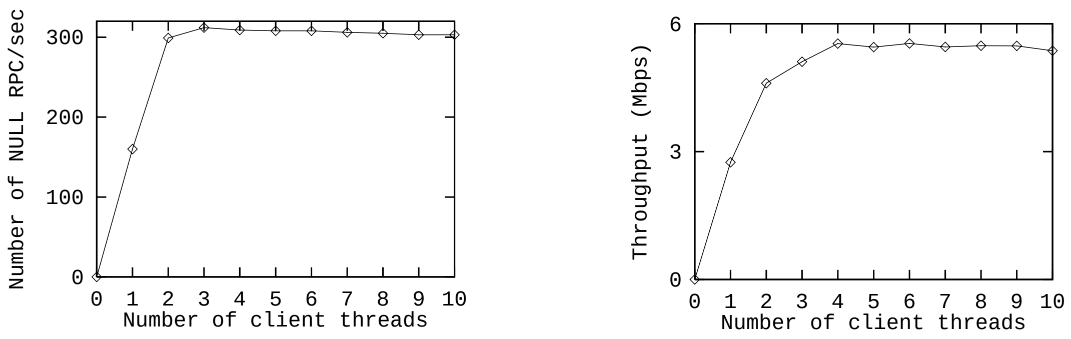

**Figure 4. Plot from Table 1.** Three threads can make 312 NULL RPCs per second. Four threads can achieve a caller-to-server throughput of 5.53 Mbps with 4016 byte data transfer per call.

## 4.2 Throughput

We measure throughput by making 1000 back-to-back RPCs with various numbers of caller threads for NULL RPC and RPC with 4016 byte request data. DCE provides user level threads for writing multithreaded client applications. DCE client and server applications are inherently multithreaded. In addition to that, multiple caller threads were employed to achieve higher throughputs. In addition to RPC performance, the throughput gives an overall assessment of the thread performance.

Table 1 and Figure 4 show that 3 threads can make 312 NULL RPC calls per second. At this level of multithreading, the server achieves its maximum utilization (as explained later). Four threads can achieve a caller-to-server throughput of 5.53 Mbps with 4016 byte data transfer per call. For comparison, a singly-threaded client is able to transfer at the rate of 6.28 Mbps with 64257 byte data transfer per call.

## 4.3 Completion time of steps

This section reports the average time to complete each step of an RPC described in section 2.1. The cost of binding and the extra overhead for the first RPC is not included in the measurement. The measurement involved instrumentation of code and generation of trace data. The overhead of instrumentation and trace generation is discussed first.

### 4.3.1 Instrumentation overhead

To our great advantage, the RS/6000s maintain a high-precision clock in a register pair. Thus we can read the clock directly using an assembly language procedure, called *curtime*, without the overhead of making a system call for timing purposes. The clock granularity is 256 nanoseconds and it takes about 440 nanoseconds to execute *curtime*. (A null procedure call takes about 200 nanoseconds.) So *curtime* is acceptable for timing events with microsecond accuracy.

Curtime was used to instrument the runtime RPC code. Because of the unavailability of the AIX kernel source code, we couldn't use *curtime* to time kernel events. Instead, the AIX kernel *trace* facility was used. Unfortunately, the overhead of *trace* is **not** insignificant compared to the granularity of our measurement. We found that every trace event generated, on an average, an overhead of 4 microseconds. All trace-derived values reported in this paper are adjusted to account for that overhead.

### 4.3.2 Hardware and system characteristics

The average time to do some basic operations and system calls are the characteristics of the machine and the operating system. The completion time of some of the operations of interest are given below. (All times are in microseconds.)

$$\begin{aligned} S \text{ bytes memory copy, } T_{memcpy}(S) &= 0.029 * S \text{ (model 530)} \\ &= 0.042 * S \text{ (model 520)} \end{aligned}$$

$$\begin{aligned} \textit{Sendmsg} \text{ with S byte data, } T_{sendmsg'}(S) &= \;\; 374 \quad \text{if } S = 0 \quad \text{(model 530)} \\ &= \left\{ \begin{array}{ll} 572 & \text{if } S = 0 \\ 747 & \text{if } S = 1392 \quad \text{(model 520)} \\ 1097 & \text{if } S = 4016 \end{array} \right. \end{aligned}$$

$$\begin{aligned} T_{sendmsg''}(S) &= \;\; 108 \quad \text{if } S = 0 \quad \text{(model 530)} \\ &= \left\{ \begin{array}{ll} 30 & \text{if } S = 0 \\ 35 & \text{if } S = 1392 \quad \text{(model 520)} \\ 309 & \text{if } S = 4016 \end{array} \right. \end{aligned}$$

$$\begin{aligned} \textit{Recvfrom} \text{ with } S \text{ bytes of data, } T_{recvfrom}(S) &= \left\{ \begin{array}{ll} 195 & \text{if } S = 0 \\ 257 & \text{if } S = 1392 \quad \text{(model 530)} \\ 438 & \text{if } S = 4016 \end{array} \right. \\ &= \;\; 254 \quad \text{if } S = 0 \quad \text{(model 520)} \end{aligned}$$

$$\begin{aligned} \text{Transmission interrupt handling, } T_{trINT} &= 176 \text{ (model 530)} \\ &= 229 \text{ (model 520)} \end{aligned}$$

Reception Interrupt handling :

Without waking up a process, $T_{recINT-nwk}$ $= 443$ (model 530)

Wake up a process - S byte data, $T_{recINT}(S)$ $= \begin{cases} 529 & \text{if } S = 0 \\ 656 & \text{if } S = 1392 \quad \text{(model 530)} \\ 714 & \text{if } S = 4016 \end{cases}$

$= 700$ if $S = 0$ (model 520)

### 4.3.3 Runtime overhead

Figures 1 and 2 show the average completion time of steps (in microseconds). When the completion time of steps are variable, *i.e.*, when they depend upon the data size, we give the value corresponding to a NULL RPC.

### 4.3.4 The physical layer

The round trip time of an RPC includes delays at the RPC layer, at the network or UDP/IP layer, and at the physical layer. The RPC and network layer delays are described in sections 4.3.2 and 4.3.3. Physical layer delay has three components: delay at the controller, transmission time, and propagation time. Propagation time, in our environment, is very small and is ignored. The delay at the controller and transmission time is estimated as follows.

Controller delay for sending S byte data is the difference between the time that a write is performed on the controller buffer and the time that the controller starts sending the packet. It is impossible for us to measure this delay without instrumenting the device driver. Because of the unavailability of the source code, we estimate the delay. We assume that a write is performed on the controller buffer when the call to the device driver routine returns (shown as the end of the time segment sendmsg$'$ in Figures 1 and 2). The time that the controller starts sending a packet is obtained by subtracting the time it takes to send the packet (*i.e.*, the network transmission time) from the time the controller interrupts after sending the packet. The controllers at the client and the server machine are assumed to be identical.

$$T_{cont-send}(S) = 326 + 0.26 * S \ \mu s \quad \text{(estimated)}$$

Controller delay for receiving data is the difference between the time that the controller receives the whole packet and the time that it issues the interrupt. We estimate this value by making a single packet RPC and measuring the total delay seen by the client between writing the request packet to the controller buffer for transmission and receiving the interrupt for the reply packet. Subtracting the delay at the server, network, and the controller sending data from the total, we get the combined reception delay at the controller on the client and the server machines. Assuming identical controllers at the client and the server machines, the controller delay for receiving the data is estimated to be the half of this combined reception delay.

$$T_{cont-recv} = 260 \ \mu s \quad \text{(estimated)}$$

$S$ bytes data transmission, $T_{trans}(S) = 0.8 * (S + 122) \ \mu s.$ [Assuming 10Mbps Ethernet]

## 5 The Round Trip Path

Not all time segments shown in Figures 1 and 2 contribute to the round trip time of an RPC. When steps in the client, server, or network overlap, the one with the largest completion time contributes to the round trip time. For example, after sending the request packet for a NULL RPC at step $C_8$ in Figure 1, the client performs steps $C_9$ through $C_{14}$. The latter steps are actually performed in parallel, while the packet is transmitted and handled by the server. In our environment, the completion time of these steps combined is far less than the time required by the network and server before the reply packet comes in, so the client blocks. In this case, the steps $C_9$ through $C_{14}$ do not affect the round trip time of the NULL RPC. Nevertheless, they consume CPU and do affect DCE performance when jobs are queued at the client. RPC steps that contribute to the round trip time constitute what we call the *round trip path* or RTP.

RTP depends on the relative speed of the client, server, and network, and may vary significantly between environments. The RTP in our experiment for an RPC that has 24096 bytes of request data is shown in Figure 5. Each vertical line segment represents the completion time of one or more steps at the client, server, network, or controller. The time is shown in microseconds next to each segment. The length of a line segment is not proportional to the time it takes to complete the corresponding steps. The RTP is shown as a solid line; steps that do not fall on the path are shown by a dotted line. The horizontal dotted arrows show the occurrence of an interrupt at the client or server.

Once we know the RTP of an RPC and the average completion time of steps, we can calculate the round trip time of that RPC by multiplying the completion time of a step that falls on the RTP, by the number of times the step is expected to be performed, and then adding all the values.

Table 2 calculates the round trip time for RPCs with various request data sizes. Column 2 lists all steps that can fall on an RTP and column 4 shows the time required to complete a step once. Calculations are shown for NULL RPC, which sends no data and produces no result, and RPCs that send data from fixed sized arrays and produce no results. A NULL RPC generates one request and one reply packet. Both IP packets are 108 bytes long. RPCs with 1392, 4016, 8032, 16064 and 24096 bytes of call data produce 1, 3, 6, 12, and 18 request packets, respectively, and a single reply packet (when no retransmission is required). Calculated round trip times are compared with the actual measured values. The error is calculated as:

$$\%\ \text{Error} = \frac{(\text{Measured time} - \text{Calculated time}) \times 100}{\text{Measured time}}.$$

## 6 Conclusion and Future Work

The base latency of inter-machine DCE RPC over UDP in our experimental setup is 6.19 milliseconds. We believe that this is low enough to provide a good basis for building distributed applications such as a distributed file system. The user level thread package also has a very low overhead and can be employed whenever there is the possibility to achieve concurrency within an application.

We now have an accurate model of where time is spent in DCE RPC. Given the size of data, our model can predict the resource requirements of an RPC — for example, the time spent at the CPU or network. We can also tell which of these time segments actually will contribute to the round trip time of the RPC. Thus, the round trip time of an RPC can also be predicted when no concurrent RPC is made.

The model in this paper does not take any queueing delays into account. Because queueing delays are sure to be present in the real world, our goal is to incorporate the measurements of this paper into a more elaborate model that does account for such delays. The resource requirements calculated from this paper can serve as the service time of jobs at various devices in a queueing network model.

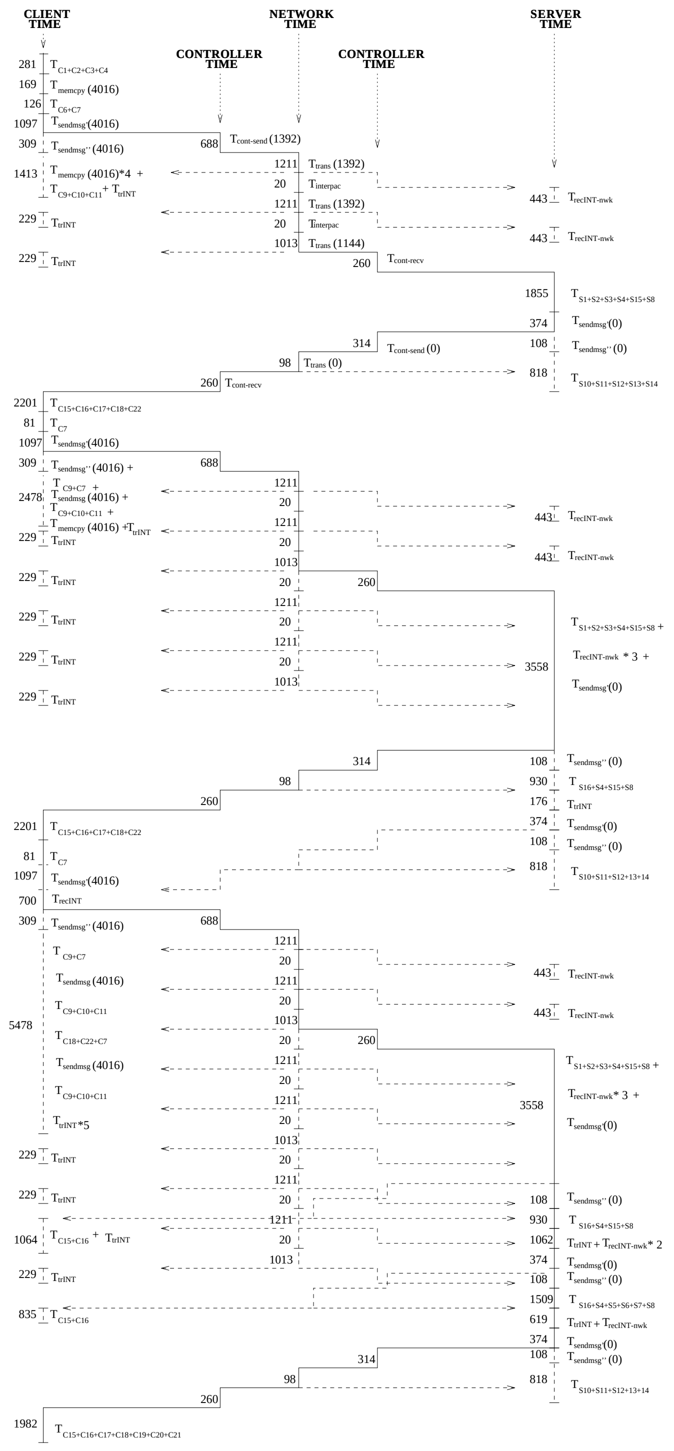

**Figure 5. The round trip path (RTP) for 24096 byte data.** Each vertical line segment represents one or more steps of the RPC. Time to complete those steps, by various devices, is shown in mircoseconds. The length of a line segment is not proportional to the time it takes to complete the corresponding steps. Steps shown with dotted lines are performed in parallel with those shown by solid lines, and do not directly contribute to the round trip time. The path indicated by solid lines shows the round trip path.

| | | | | No. of times the step is performed | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Machine/ | Step | Completion time | Unit cost | NULL | RPC with request data (byte) | | | | |
| Device | | (if variable) | | RPC | 1392 | 4016 | 8032 | 16064 | 24096 |
| Client | $C_{1+2+3+4}$ | | 281 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $C_5$ | $T_{memcpy}(0)$ | 0 | 1 | | | | | |
| | | $T_{memcpy}(1392)$ | 58 | | 1 | | | | |
| | | $T_{memcpy}(4016)$ | 169 | | | 1 | 1 | 1 | 1 |
| | $C_6$ | | 45 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $C_7$ | | 81 | 1 | 1 | 1 | 2 | 3 | 3 |
| | $C_8$ | $T_{sendmsg'}(0)$ | 572 | 1 | | | | | |
| | | $T_{sendmsg'}(1392)$ | 747 | | 1 | | | | |
| | | $T_{sendmsg'}(4016)$ | 1097 | | | 1 | 2 | 3 | 3 |
| | $C_{12}$ | $T_{trINT}$ | 229 | | | | | | |
| | $C_{15}$ | $T_{recINT}$ | 700 | | | | | 1 | 1 |
| Controller | | $T_{cont-send}(0)$ | 326 | 1 | | | | | |
| | | $T_{cont-send}(1392)$ | 688 | | 1 | 1 | 2 | 3 | 3 |
| Network | | $T_{trans}(0)$ | 98 | 1 | | | | | |
| | | $T_{trans}(1392)$ | 1211 | | 1 | 2 | 4 | 6 | 6 |
| | | $T_{trans}(1144)$ | 1013 | | | 1 | 2 | 3 | 3 |
| | | $T_{interpac}$ | 20 | | | 2 | 4 | 6 | 6 |
| Controller | | $T_{cont-recv}$ | 260 | 1 | 1 | 1 | 2 | 3 | 3 |
| Server | $S_1$ | $T_{recINT}(0)$ | 529 | 1 | | | | | |
| | | $T_{recINT}(1392)$ | 656 | | 1 | | | | |
| | | $T_{recINT}(4016)$ | 714 | | | 1 | 2 | 3 | 3 |
| | | $T_{recINT-nwk}$ | 443 | | | | | 3 | 9 |
| | $S_{2+3}$ | | 364 | 1 | 1 | 1 | | | 3 |
| | $S_4$ | $T_{recvfrom}(0)$ | 195 | 1 | | | | | |
| | | $T_{recvfrom}(1392)$ | 257 | | 1 | | | | |
| | | $T_{recvfrom}(4016)$ | 438 | | | 1 | 2 | 3 | 5 |
| | $S_{5+6+7}$ | | 614 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $S_{15}$ | | 278 | | | | 1 | 2 | 4 |
| | $S_{16}$ | | 396 | | | | | 1 | 2 |
| | $S_8$ | | 61 | 1 | 1 | 1 | 2 | 4 | 5 |
| | $S_9$ | $T_{sendmsg'}(0)$ | 374 | 1 | 1 | 1 | 2 | 3 | 3 |
| | | $T_{sendmsg}(0)$ | 482 | | | | | 1 | 2 |
| | $S_{13}$ | $T_{trINT}$ | 176 | | | | | | 2 |
| Controller | | $T_{cont-send}(0)$ | 326 | 1 | 1 | 1 | 2 | 3 | 3 |
| Network | | $T_{trans}(0)$ | 98 | 1 | 1 | 1 | 2 | 3 | 3 |
| Controller | | $T_{cont-recv}$ | 260 | 1 | 1 | 1 | 2 | 3 | 3 |
| Client | $C_{15}$ | $T_{recINT}$ | 700 | 1 | 1 | 1 | 2 | 3 | 3 |
| | $C_{16+17}$ | | 443 | 1 | 1 | 1 | 2 | 3 | 3 |
| | $C_{18}$ | $T_{recvfrom}(0)$ | 254 | 1 | 1 | 1 | 2 | 3 | 3 |
| | $C_{19+20+21}$ | | 585 | 1 | 1 | 1 | 1 | 1 | 1 |
| | $C_{22}$ | | 804 | | | | 1 | 2 | 2 |
| Summation of segments in RTP | | | | 6454 | 8351 | 11315 | 22018 | 32691 | 41070 |
| Measured round trip time | | | | 6169 | 8490 | 11414 | 21686 | 34881 | 44853 |
| %Error | | | | -4.62 | 1.64 | 0.87 | -1.53 | 6.28 | 8.43 |

**Table 2. Calculation of round trip time of RPCs.** Round-trip time is the sum of the completion time of steps along the round trip path. All times are in microseconds.

## Acknowledgements

We thank William Andros Adamson for help instrumenting the DCE RPC code and Sarr Blumson for his critical review of this paper. Special thanks to Mary Jane Northrop for editing. This work was supported by a research partnership with IBM.

## References

[1] Balakrishnan Dasarathy, Khalid Khilily, and David E. Ruddock. "Some DCE Performance Analysis Results". In *Proceedings, International DCE Workshop, University of Karlsruhe, Germany*, 1993. Published as Lecture Notes in Computer Science, No. 731, A. Schill (Ed.), Springer-Verlag, 1993.

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

[3] Rolf Rabenseifner and Armin Schuch. "Comparison of DCE RPC, DFN-RPC, ONC and PVM". *Proceedings, DCE - The OSF Distributed Computing Environment, International DCE Workshop, Karlsruhe, Germany*, pages 39–46, October 1993.

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

[5] Mendel Rosenblum. "The Performance of Sun's Remote Procedure Call". Technical report, CS 266, University of California, Berkeley, 1986.

[6] Michael D. Schroeder and Michael Burrows. "Performance of Firefly RPC". *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.