Using Shim Technology to Monitor DCE Runtime Performance

Gregory M. Oster and Richard B. Bunt

Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada S7N 5A9
{oster, bunt}@cs.usask.ca

Abstract

Obtaining performance data for application or system software is typically difficult, especially when the source code is not available. While popular techniques such as event trapping or monitoring external network traffic may provide some indication of performance, their various shortcomings restrict the domain of applicability. In this paper, we illustrate a new approach to obtaining performance data that appears to address some of these concerns and demonstrate it in the context of DCE applications and system services. This approach is architecture-independent, does not rely on application or system services source code, and is easy to use.

1 Introduction

Obtaining good performance data is always a challenge, but for packaged system and application software it is especially difficult. Since such software rarely comes with built-in mechanisms for monitoring its performance, a variety of methods are employed to obtain performance data. Where source code for the software is available, a typical approach is to hand-instrument the application to provide the required data. This work is often tedious, and requires detailed knowledge of the internal workings of the software and hand-tuning of the instrumentation code in order to get the desired data. Since such software does not usually come with source code, however, other methods must often be employed. Techniques based on external observations of network activity are widely used in distributed environments, but such techniques cannot provide a detailed breakdown of the internal workings of the software and so are unable to provide an indication as to what components of the software are responsible for the observed problems with its performance. We have been exploring yet another approach, which we call the *Measurement Shim*. In this paper, we illustrate the application of Shim Technology to the task of monitoring the runtime performance of a piece of commercial software.

DCE[1] system services and applications usually come without source code, and therefore must usually be monitored by external techniques. In-house application source can be hand-instrumented, but the effort required to do so is typically quite large. System services, such as those provided by the various servers in a DCE cell, can typically be monitored only using external techniques, such as observing the traffic at the network layer, as was done in several recent performance investigations [7, 8]. This approach, however, has some shortcomings. For example, where a server does not generate network traffic, such as when it is communicating with a client running on the same machine, obtaining performance data for that

[1] The Open Software Foundation's Distributed Computing Environment [6] provides tools and services to support the development and operation of distributed systems and applications. It is available commercially on a number of popular computing platforms.

server may be nearly impossible using the network monitoring technique.

Shim Technology provides a framework for exploiting well-defined interfaces between two software components. The Shim described in this paper not only allows the monitoring of applications and system services within a DCE framework, but does so without requiring source code, or even a re-linking of object code. It provides reliable, accurate data quickly and effectively.

The rest of this paper is organized as follows. Section 2 gives a brief discussion of the merits and deficiencies of some of the current approaches to measuring performance, with specific reference to DCE applications and services. Section 3 briefly outlines the Shim approach. Section 4 discusses the actual construction of our DCE Measurement Shim, and Section 5 describes our experiences with it. The discussion assumes a small amount of familiarity with dynamically linked applications and the run-time linkage process performed by a dynamic loader. Section 6 examines why the Shim technique works so well both in general and with respect to measuring DCE Performance. A summary of Shim benefits and directions for future work are presented in Section 7.

2 Current Approaches to Measuring DCE Performance

There are a number of techniques currently used to measure the performance of DCE applications and system services:

1. Hand instrumentation of the application. The primary advantage of this approach (used, for example, by Khander *et al.* [5]) is that the instrumentation can be very specific in the data it collects and very light-weight. The primary disadvantages are that it requires application source code to be available, it requires the programmer to code specifically to the application, and instrumenting some functions (such as the marshalling/unmarshalling routines) would require editing the stub code generated by the IDL compiler[2]. It may also be necessary to duplicate the instrumentation code for each of the functions under observation, as these function calls may take place in a number of places within the application.

2. Automated instrumentation of the application. In an application development tool, instrumentation routines could be automatically inserted into the application source code. As discussed by Blumson *et al.* [2], automatically generating instrumentation code solves many of the problems of hand instrumentation. In their work the IDL compiler is used to insert the necessary instrumentation code into the stubs automatically. While this approach may be acceptable where source code is available, it cannot be used where source for the applications or system services is unavailable. This approach also requires access to the specialized IDL compiler or development tool.

3. Network monitoring. Recent work [7, 8] has shown how monitoring the applications and system services at the network level can provide a great deal of useful information, without requiring application or system services source code. The primary drawback, however, is that only cross-network activities can be seen. Obtaining performance data in this fashion also has more room for error, as there is no way to account for things like the amount of time spent marshalling or unmarshalling the data, or for the time required for the data to move up or down the various TCP or UDP protocol stacks. Thus a method of instrumentation which is somewhat "closer" to the application client or server is more desirable.

4. Instrumentation of the run-time library. This approach, which was used by Friedrich *et al.* [4], is quite similar to the

[2] The IDL (Interface Definition Language) compiler generates the application client and application server stub code from an interface specification. This specification, which is written in the Interface Definition Language, completely describes the function exported by the application server.

approach used in the DCE Measurement Shim. The primary difference is that they add the instrumentation directly to the run-time library, and thus source code for the run-time library is required. In the DCE Measurement Shim, the instrumentation is added to a new interface to the run-time library, and the run-time library source is not required. To create the Shim, however, a complete specification is required for all the functions provided by the run-time library.

As in all these approaches, the ability to gather large amounts of data is also present in the DCE Measurement Shim.

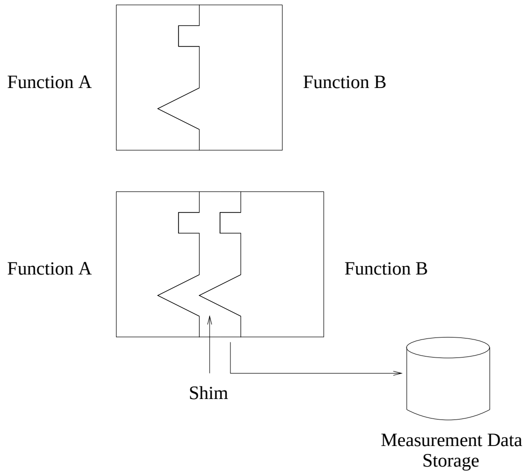

Figure 1: Inserting the Shim.

3 Shim Technology

A "shim" is a thin piece of material inserted between two surfaces, often for the purpose of evening or leveling. The notions that the Shim is small, coupled tightly to its surrounding components, and easy to use are central to the goals of Shim Technology.

In well-written software all functions communicate through a well-defined interface. This well-defined interface is completely described in terms of the types of data sent or received across the interface. Any such well-defined interface between two components of a software system can be exploited using a Shim.

Figure 1 illustrates the interposition of a Shim between two functions, A and B. All communication between A and B now passes through the Shim. To the *calling function* A, the Shim interface looks and behaves exactly the same as the interface of the *called function* B. To the called function, the Shim interface looks and behaves exactly the same as the calling function. The Shim intercepts the function calls, passes argument data to the called function, and passes the results back to the calling function. At the same time, Shim code collects data on the communications that pass through it. Figure 1 shows the interface between Function A and Function B, and the position of the Shim.

The Shim is designed to be light-weight and compact, and to impede the movement of data as little as possible. We have applied Shim Technology successfully to several performance measurement tasks, one of which is a study of DCE RPC performance.

4 Building the DCE Measurement Shim

4.1 Objectives of the DCE Measurement Shim

DCE provides a good vehicle for this demonstration because it is a large piece of commercial software that comes without source code or measurement facilities. Our focus for this presentation is on DCE RPC performance. In particular, the round-trip time (RTT) for RPC requests and the RPC overhead (e.g. for the first RPC) were identified as good candidates for observation.

As the basis for client-server communication in DCE, RPCs are fundamental to the operation of DCE system services and applications. Providing insight into what factors affect RPC performance is a primary goal of the DCE performance research at the University of Saskatchewan. In the work discussed in [1, 7, 8], a network packet sniffer (`tcpdump`) was used to obtain traces of the RPC traffic generated by the DCE application and system services under observation on an experimental DCE testbed. These traces were then analysed by custom-written scripts. As mentioned in Section 2, this approach has a number of problems. For example, the details of packet processing (e.g. marshalling/unmarshalling, TCP or UDP protocol processing, etc.) remain hidden, as do communications which occur on a single machine. As well, even though the tracing is non-intrusive it is often difficult for the tracing machine to capture all of the packet contents needed for the analysis, and the lost packets introduce a sometimes significant source of error. The application of the Shim Technology to the problem seems to address these concerns.

4.2 A Brief Background on Shared Libraries in DCE

In the OSF DCE version 1.3 used in this work, all the executables are dynamically linked, and most rely on the shared library `libdce.so`. This library contains all the primary functions that DCE uses, including "private" ones which are typically used only by the stub code (generated by the IDL compiler). At run-time, a given application will use the shared library to satisfy any unresolved bindings.[3]

4.3 The Structure of the Shim

The DCE Measurement Shim, which is contained in a library called `libdceshim.so`, relies on the way run-time bindings are handled by the operating system linker/loader. The DCE Measurement Shim exports exactly the same function interfaces as the `libdce.so` library, and calls the original functions from the `libdce.so` library. The difference is that the functions in `libdceshim.so` have an opportunity to obtain some performance data on that particular function, both before and after the "real" function is called.

To illustrate the structure of the DCE Measurement Shim, suppose `rpc_function()` is a function in `libdce.so`. The function `shim_rpc_function()` is created, and it both calls `rpc_function()` and obtains the appropriate performance metrics for that call. A (proto-)typical implementation is shown in Figure 2.

When `libdceshim.so` is loaded, there is a small amount of initialization which must take place. In particular, the `init_shim()` function must be called as soon as `libdceshim.so` is loaded. This function, which is shown (in part) in Figure 3, is responsible for loading the "real" `libdce.so` (which in our DCE installation is found in `/usr/opt/DCE130/usr/shlib`), and for creating the function hooks which will be used to call the "real" DCE functions. In the prototype implementation of the DCE Measurement Shim, the "`-init init_shim`" option was passed to the linker `ld`.[4] This option requests that the `init_shim()` function be called immediately after the library is loaded.

4.4 The Application's Viewpoint

From the perspective of the application, nothing has changed. There are no additional hooks or special requirements needed to use the DCE Measurement Shim. All function calls take place in exactly the same manner as before. The only change is that the function calls now take place through the Shim mechanism.

Since `libdceshim.so` exports the same functions that `libdce.so` exports[5] it is possible to create a "drop-in" replacement for `libdce.so`. The technique for doing this is as follows:

1. Compile a version of `libdceshim.so` with the path to the original `libdce.so` well specified.
2. Rename `libdceshim.so` to `libdce.so`, without removing the original `libdce.so`.
3. Change `LD_LIBRARY_PATH` so that when the run-time loader searches for `libdce.so` it will find the new one first.

The current version of `libdceshim.so` is capable of running most locally compiled programs using the above technique. As well, both

[3] The "binding" here refers to the function binding performed by the linker/loader. It does not refer to the endpoint binding in DCE.

[4] This option does not appear to be common in most implementations of `ld` on other platforms. Only a trivial code change is required to get the same behaviour if the "`-init init_shim`" option is not available.

[5] The current implementation exports only a limited subset since it is only a prototype for demonstration purposes.

```
void rpc_call_transceive(
     rpc_call_handle_t call_handle,
     rpc_iovector_p_t in_call_args,
     rpc_iovector_elt_t *out_call_args,
     ndr_format_t *remote_ndr_fmt,
     unsigned32 *status )
{
     struct timeval t1,t2;
     gettimeofday(&t1,NULL);
     printf("Start %s :  %.9f\n", "rpc_call_transceive",
          (double) t1.tv_sec + (double) t1.tv_usec / 1000000 );
     shim_rpc_call_transceive( call_handle, in_call_args, out_call_args,
          remote_ndr_fmt, status);
     gettimeofday(&t2,NULL);
     printf("End %s :  %.9f\n", "rpc_call_transceive",
          (double) t2.tv_sec + (double) t2.tv_usec / 1000000 );
}
```

Figure 2: Source Code for the Shim Version of `rpc_call_transceive()`.

```
void (*shim_rpc_call_transceive)();
void *dl_handle;

void init_shim()
{
     dl_handle = dlopen("/usr/opt/DCE130/usr/shlib/libdce.so",RTLD_NOW);
     if (dl_handle) printf("Using DCE Shim library.\n");
     else printf("DCE Shim library error:  %s\n",dlerror());
     shim_rpc_call_transceive=dlsym(dl_handle,"rpc_call_transceive");
}
```

Figure 3: Source Code for the Shim Initialization.

rpccp and sec_admin (both are pre-compiled components of the OSF DCE distribution, and come without source code) can be instrumented this way.

5 Experiences with the DCE Measurement Shim

In order to gauge the effectiveness and usefulness of the DCE Measurement Shim two simple experiments were performed. The primary focus of these trials was to ensure that the implementation of the Shim would be both practical and useful. A number of studies have reported results on DCE RPC performance [3, 5], including some of our own [7, 8]. Among issues of interest are the round-trip time (RTT) for RPC requests, and the overhead associated with the first RPC. In this section we discuss the experiments performed and the results obtained, and illustrate the advantages of the Shim Technology in the measurement of RTTs and the investigation of the first RPC.

5.1 The Experimental Environment

The DCE experimental environment for the Shim testing consisted of a single DCE cell containing a number of machines running Digital UNIX and OS/2. For these experiments, two DEC Alpha 2100A-500MPs were used. Each machine had four 190MHz Alpha AXP21064 CPU's and 256MB of main memory, and ran Digital UNIX with OSF's DCE 1.3. These machines were connected via a 10Mbps Ethernet. The network monitoring machine (running tcpdump) used for the experiments was a DEC Alpha 3000/300.

The application used in the experiments is a simple "data exchange" application written to move blocks of data between the application client and the application server. Explicit (string) binding[6] was used in the application, and the application used the UDP transport protocol. The data transferred was simple NULL-terminated character strings of various lengths. The data transfered in each RPC was a multiple of 2K, ranging from 0 to 64K.

Output from the Shim was simply printed to the standard output, and captured to a file for offline analysis. Collecting data in this fashion was deemed reasonable as the implementation was done primarily as a "proof of concept". Both tcpdump and the Shim were used simultaneously, so that both measurement techniques were applied to the same client-server interaction. Since the Shim was used only with the application client, the existence of the Shim would not interfere with the data captured at the network level, and since tcpdump is a passive monitor, the Shim would not be affected by tcpdump. Simple analysis tools post-processed the traces collected and summarized the results.

5.2 The Experiments

The focus of the first experiment was to investigate the ability of the DCE Measurement Shim to obtain performance data on some of the more commonly used RPC-related functions within DCE. The goal was to determine that this data collection was both feasible, and easy to perform. The RPC-related functions which were monitored in this experiment are used only in the DCE stubs. These functions, which are listed in the first column of Table 1, are normally instrumentable only by hand-instrumenting the stub source code that is generated by the IDL compiler. The functions rpc_ss_ndr_marsh_interp() and rpc_ss_ndr_unmar_interp() handle the marshalling and unmarshalling, respectively, of all data sent to or from the client or server. The function rpc_call_start() is responsible for setting up the connection during the first RPC. rpc_call_transceive() is the function in the client stub code which actually initiates the data transfer from the client to the server.

Table 1 summarizes the data values obtained in the first experiment. The values shown here will be slightly larger than the actual values, given the simple fashion in which printing of the timestamps was done (See Figure 2). Printing the timestamps before and after each call, however, provides a very informative trace of the sequence of function calls. The perfor-

[6] Not to be confused with the library binding discussed in Section 4.2.

mance values provide interesting insight into the application being monitored and the hardware the application is being run on. For example, the amount of time spent marshalling or unmarshalling data is quite small (0.2229 and 0.1035 milliseconds, respectively), which indicates that the data needs very little transformation (if any) to conform to the network data representation.

The goal in the first experiment of easily collecting the performance data has been achieved. Not only is the DCE Measurement Shim easy to use, but it can also provide performance data on functions which previously could have been obtained only via hand-instrumentation.

The goal of the second experiment was to compare the accuracy of the performance results obtained from the DCE Measurement Shim with those obtained from the `tcpdump` trace. In this experiment, the "exchange" application was used to send 10000 RPCs from the client to the server. Both the DCE Measurement Shim and `tcpdump` simultaneously monitored the exchange of data. In this experiment, the Shim printed the RTT results only after the `rpc_call_transceive()` function (the only instrumented function in this experiment) completed, and thus the overhead of the Shim is quite small. Figure 4 shows the source code used for the `rpc_call_transceive()` function used in this experiment. Note that the RPC data size from the Shim (`total_size` in Figure 4) is slightly smaller than the RPC data size determined from the `tcpdump` trace. This is because the data is only beginning to go through the various protocol stacks, and does not yet have all the appropriate headers attached.

Figure 5 shows the average round-trip time (RTT) performance of the application, for each RPC packet size and for each data capture method. Figure 6 shows the best (i.e. minimum) RTT performance of the application, again for each RPC data size and each capture method. The difference between the measurements taken by the two capture methods reflects the amount of time required to do the protocol processing.

The difficulties in obtaining accurate performance measurements from the network via `tcpdump` are visible in the spikes of both of the Figures. These spikes are likely due to bursts of network traffic which occurred while the experiments were being performed. In this trace, about 5% of the packets were not captured, primarily because the machine running `tcpdump` was unable to keep up with the network traffic when the bursts occurred. Lost packets are difficult to account for in the analysis software, and constitute a source of error in the experiment. As the figures show, the performance data returned by the DCE Measurement Shim does not suffer from the difficulties associated with packet loss. As these results indicate, the Shim can be used to measure application performance even when the network environment cannot be easily controlled.

The first DCE RPC is known to take longer than subsequent RPCs, due to the initial RPC binding call which takes place during the first RPC. To this point, measuring the exact time required in actually performing the first real data transmission in the first RPC has been quite difficult, since there was no easy way to discern what part of the transfer was the connection setup time, and what part was the data transmission. Thus, the first RPC is typically ignored, as it is in the scripts which analyse the traces obtained from `tcpdump`. In the data returned by the Shim, the additional work performed by the binding process would be evidenced by a spike in Figure 5 at the 132 byte position. The spike is not shown in this plot as the data points it represents are not part of the actual "exchange" of application data. The spike, to a value of 21.233 seconds (average), is due to three calls to `rpc_call_transceive()`. The first two calls, at 31.99 and 31.69 seconds each, failed to locate an appropriate application server. The third call, which only took 0.02 seconds, successfully located a server and completed the binding. With the DCE Measurement Shim, the various binding calls were easily separated from the actual data transmission performed in the first RPC, and thus the actual data transmission time for the first RPC could actually be measured.

Function Monitored	Mean Execution Time (msec)	Variance (msec)
rpc_ss_ndr_marsh_interp()	0.2229	0.0062
rpc_ss_ndr_unmar_interp()	0.1035	0.0017
rpc_call_start()	0.2477	0.0036
rpc_call_transceive()	64.7599	23.3057

Table 1: Initial Results From the DCE Measurement Shim.

```
void rpc_call_transceive(
     rpc_call_handle_t call_handle,
     rpc_iovector_p_t in_call_args,
     rpc_iovector_elt_t *out_call_args,
     ndr_format_t *remote_ndr_fmt,
     unsigned32 *status )
{
     int i;
     int total_size=0;
     struct timeval t1,t2;
     gettimeofday(&t1,NULL);
     for(i=0;i<in_call_args->num_elt;i++) {
          total_size+=in_call_args->elt[i].data_len;
     }
     shim_rpc_call_transceive( call_handle, in_call_args, out_call_args,
          remote_ndr_fmt, status);
     gettimeofday(&t2,NULL);
     printf("Totals:  %s %d %.9f\n", "rpc_call_transceive", total_size,
          ((double) t2.tv_sec + (double) t2.tv_usec / 1000000) -
          ((double) t1.tv_sec + (double) t1.tv_usec / 1000000) );
}
```

Figure 4: Source Code for rpc_call_transceive() as Used in the Second Experiment.

5.3 Other Results

For all experiments, the library libdceshim.so was used as a direct replacement for libdce.so, using the technique described in Section 4.4. This shows how easily the Shim can be used, and how transparent the Shim is to the distributed application. The ease with which an application can be instrumented by simply changing the library search path (as outlined in Section 4.4) was demonstrated when a local graduate student's DCE application was instrumented without requiring a re-compile of the application.

The "call tracing" capabilities of the Shim have also proven useful, although this functionality has not been fully explored. With the display of simple "Start" and "End" timestamps before and after a few selected functions (See Figure 2), it became apparent quickly which DCE functions were being called, and in what order, and how that behaviour related to the functioning of the DCE application. Such tracing was previously possible only by hand instrumenting the application stub code, or the DCE Runtime library. For debugging applications and servers, a call-level trace provides valuable assistance in examining the behaviour of the application. Again, the use of the DCE Measurement Shim with a graduate student's application provided immediate insight into how the "hidden" functions of the application behaved when the application was running.

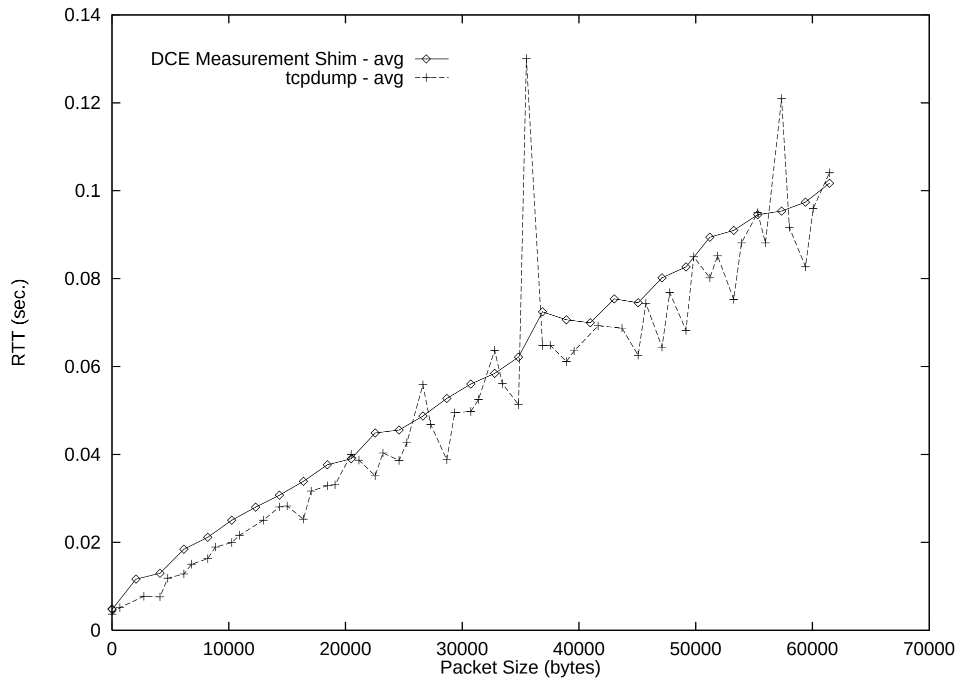

Figure 5: Average RPC Round-Trip Times as a Function of Packet Size.

5.4 Overhead of the DCE Measurement Shim

All the instrumented functions in the DCE Measurement Shim will have a small performance penalty introduced by the instrumentation. If a function simply has a "pass through" Shim defined for it, then the additional cost induced by the Shim is negligible. When function `rpc_call_transceive()` is called, the function `shim_rpc_call_transceive()` is called, and `shim_rpc_call_transceive()` is really the original `rpc_call_transceive()` exported by `libdce.so`. Thus the function's parameters must be passed one additional time. How expensive this passing is will depend on the nature of the data being exchanged, and how efficiently a given architecture can do the parameter passing. On the machines used in these experiments, the "cost" of adding the instrumentation was about 1 millisecond per function call, including the cost of `printf()`'s, for a call similar to `rpc_call_transceive()`. Without the `printf()`'s, 1 billion such calls can be done in 64 seconds, a cost of only 0.064 microseconds per call.

If the Shim for a particular function is doing more than just passing data through it, then there is a penalty paid for the amount of performance data collected, and a penalty for how that performance data is returned to the user.

6 Why this works so well

While the measurement Shim is presented here in the context of DCE performance measurement, the technique is applicable to many other domains. Since the focus of this technique is the exploitation of well-defined interfaces, where any such interface exists the flow of data across the interface can be measured.

This measurement technique results in no loss of data, save for a lack of sufficient space for storing the output. There is also minimal interference with a running system – the overhead of the Shim is directly related to the amount of monitoring or measurement being done on the interface.

Although this technique may seem at first glance to be "yet another data acquisition technique", it is important to realize that the use-

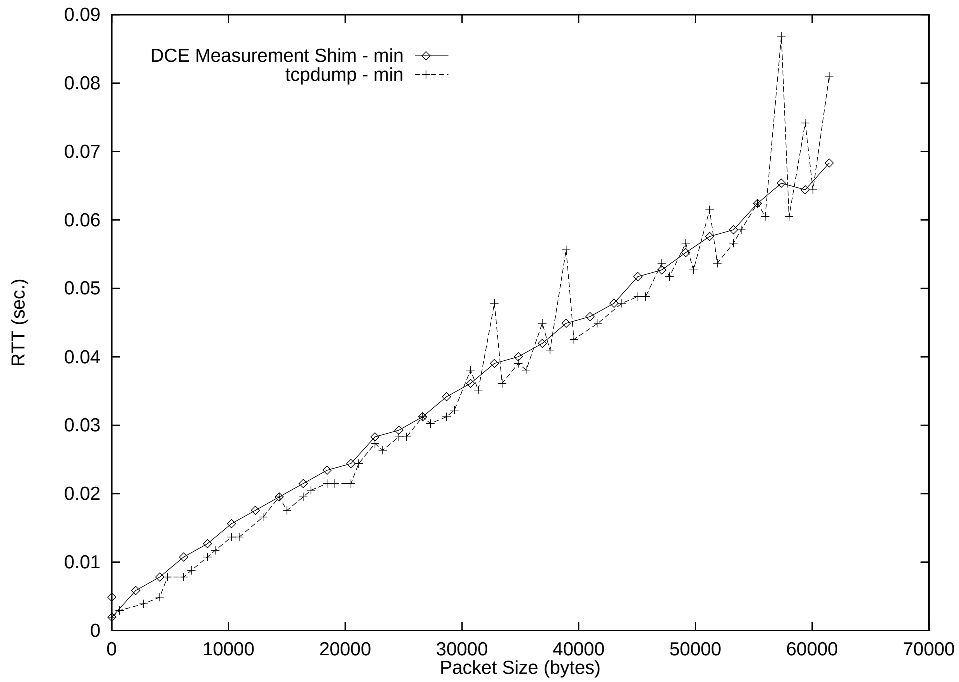

Figure 6: Minimum RPC Round-Trip Times as a Function of Packet Size.

fullness of this technique extends to all interfaces where those interfaces are well-defined. In the age of objects and methods, and API specifications, most interfaces between components of systems are well-defined, and thus would be well-suited for being monitored or measured via a Shim.

What makes the Shim so useful in this particular setting are the following factors:

- Most of the DCE function interfaces are very well-defined. For example, all the standard DCE function interfaces are well-documented. For a number of the hidden functions, like `rpc_call_transceive()`, sufficient information was available from the various header files and other examples to make some instrumentation of these functions possible. Where a function prototype is unavailable, however, the construction of a Shim for that function is very difficult if not impossible.

- The shared library implementation in Digital UNIX allows easy and "on-the-fly" changing of the library search path. This means that not only is it fairly arbitrary where the Shim is located, but it is also possible to have a number of different Shims, and easily switch between them. Since each user is able to define his/her own library search paths, different users may be using different Shims on their individual applications.

- The executable programs are not tightly coupled to the libraries. Other operating systems with shared libraries require those libraries to be in a place that is specified when the executable is compiled. While this ensures that the correct library revision is always used, it makes using the Shim technique in this fashion more difficult. In this situation, the original library must be replaced with the Shim, and the original library moved to some new location where the Shim can access it. This requires that the new Shim export all of the interfaces that the original library does – a subset of the interfaces will not be sufficient to support all of the executables supported by the library. It also makes it very difficult to switch between instrumenting and not instrumenting applications, as to perform such library changes would often require administrator privileges. As each executable is expecting to find the libraries in only a single place, this also has the drawback of forcing all users to either use the same library, be it one of any number of instrumented versions, or the non-instrumented version.

7 Future Work with the DCE Measurement Shim

At the time of writing this paper, only a very preliminary Measurement Shim prototype has been developed for DCE. Among planned extensions are the following:

- Collecting data other than timestamps (e.g. usage counts, number of bytes sent, data particular to the given function, etc.).
- Instrumenting all of the DCE functions. Currently there are around 560 DCE functions for which performance data could be collected.
- Using a centralized repository, for example a DCE application server which collects performance data and stores it in a central location.
- Extending the implementation to allow for "quiet" data collection, where the data is saved in the Shim for later retrieval. This should significantly reduce the (already small) overhead introduced by adding the Shim.
- Incorporating the DCE Measurement Shim into a real-time DCE performance monitor [1].
- Examining the "call tracing" capabilities of the Shim, especially with respect to gaining a greater understanding of how the internals of the DCE runtime libraries are used.
- Extending the DCE Measurement Shim idea to obtain performance data for other shared libraries.

8 Conclusion

Shim Technology offers an elegant and effective method of capturing performance data for each of the functions exported by the DCE runtime library. The prototype implementation demonstrates that the technique works well for obtaining performance information. The approach achieves the goals of being architecture-independent, not reliant on application or system services source code, and easy to use.

In its initial application, the use of the DCE Measurement Shim has allowed us to identify which parts of the first RPC are taking the additional time and provides a more stable picture of round-trip time performance. Initial experimentation indicates that with a simple relinking, this new library can be used with any DCE application. We are encouraged by our success in this application of Shim Technology and are investigating its use in other performance investigations. The technique should be applicable to any well-defined interface.

9 Acknowledgements

We gratefully acknowledge the contributions to this work of Doug Konkin, to whom we credit the original Shim concept. Funding for this research has come from the Natural Sciences and Engineering Research Council of Canada (grants CAP171120 and OGP0003707) and from IBM Canada's Centre for Advanced Studies as part of the Mandas project.

References

[1] M. A. Bauer, J. P. Black, R. B. Bunt, A. El Rayess, P. J. Finnigan, T. Kunz, H. L. Lutfiyya, A. D. Marshall, P. Martin, G. M. Oster, W. Powley, J. Rolia, D. Taylor, and M. Woodside, "Managing Distributed Applications and Systems: An Architectural Experiment", *IBM Systems Journal*, to appear.

[2] S. Blumson, M. Carter, and D. Hyde, "Automatic Insertion of Performance Instrumentation in Distributed Applications", CITI Report, University of Michigan, 1995.

[3] C. Corn, T. Li, R. Pekowski, and B. Santeford, "Performance of Key Functions in DCE for OS/2", *Personal Systems*, January/February 1994, pp. 60-70.

[4] R. Friedrich, J. Martinka, T. Sienknecht, and S. Saunders, *Integration of Performance Measurement and Modeling for Open Distributed Processing*, Hewlett-Packard Laboratories, Palo Alto, California 94304 USA, 1994.

[5] A. M. Khandker, P. Honeyman, and T. Teorey, "Performance of DCE RPC", *Proc. 2nd International Conference on Services in Distributed and Networked Environments*, Whistler, BC, June 1995, pp. 2-10.

[6] Open Software Foundation, *Introduction to OSF DCE*, Prentice Hall, 1992.

[7] Y. Sun, *Measuring and Modelling RPC Performance in OSF DCE*, M.Sc. Thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Canada, October 1997 (expected).

[8] Y. Sun, R. B. Bunt, and G. M. Oster, "Measuring RPC Traffic in an OS/2 DCE Environment", *Proc. 5th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 97)*, Haifa, Israel, January 1997, pp. 51-54.