CITI Technical Report 95–4

Automatic Insertion of Performance Instrumentation for Distributed Applications

Sarr Blumson
sarr@citi.umich.edu

Mark Carter
markc@citi.umich.edu

Daniel Hyde
drh@citi.umich.edu

ABSTRACT

The Open Software Foundation's Distributed Computing Environment (DCE) is based on a remote procedure call (RPC) paradigm. This paradigm provides a convenient, simplifying method for building distributed applications, but the simplification often makes performance tuning more difficult by concealing network operations from the programmer. To help evaluate the performance of distributed applications, we modified the Interface Definition Language (IDL) compiler of DCE RPC to automatically insert performance instrumentation.

February 7, 1995

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Automatic Insertion of Performance Instrumentation for Distributed Applications

Sarr Blumson, Mark Carter, and
Daniel Hyde

February 7, 1995

1. Introduction

The Center for Information Technology Integration (CITI) is the research and advanced development organization within the Information Technology Division of the University of Michigan. A important part of CITI's charter is to identify and help introduce significant new technologies to the University.

One such technology is the Distributed Computing Environment (DCE) of the Open Software Foundation, an industry consortium that includes a number of hardware and software vendors and end users [1]. DCE provides a coherent and integrated solution to many of the problems involved in building distributed computing systems. At the same time, however, it is a relatively new technology, and lacks many of the details that make a mature and robust product. Much of our work over the last several years has involved filling some of these gaps.

One area of weakness in DCE is in performance monitoring. Distributed computing introduces new complications to performance measurement; the performance of a program may be affected by events on a number of different machines and many operations that appear to be simple function calls may actually involve communication with other machines. The work described in this paper focuses on the second part: instruments that separate the performance of a distributed program into the performance of its individual components.

Section 2 describes DCE at a level of detail sufficient to describe our instrumentation scheme. Section 3 motivates and outlines our specific approach, while section 4 describes some related work. Section 5 describes the data collection environment in which we were working. Section 6 presents our implementation in more detail and focuses on some of our specific design decisions. Section 7 explores the things we learned by this exercise, while section 8 offers some conclusions and suggestions for future work.

2. Brief Background on DCE

The Distributed Computing Environment (DCE) is a collection of standard interfaces and supporting software for creating secure, robust distributed applications that can interoperate without regard to the particular platforms on which they are running [1]. This standard infrastructure that

DCE provides is available on a variety of hardware platforms, from personal computers to mainframes.

DCE is a client/server system based on a remote procedure call (RPC) [2] paradigm. The RPC mechanism is the center of DCE (Figure 1); a variety of other services (such as naming and security) support the RPC mechanism and are built on it.

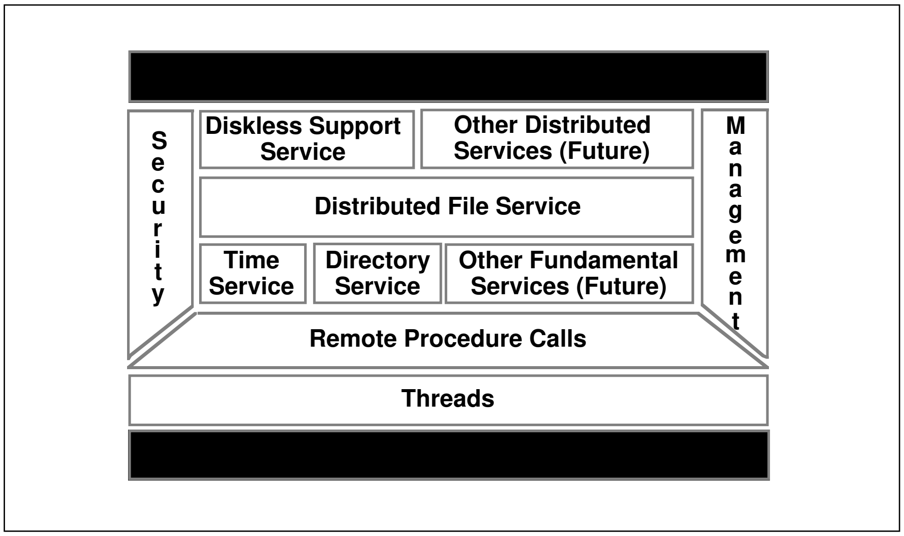

Figure 1. DCE Architecture

In DCE, applications are developed using an interface definition language (IDL) that describes interfaces (collections of procedures) that are callable across a network. An IDL compiler automatically constructs stub procedures, which create an illusion that the procedure and the caller are actually in the same address space, Figure 2. These stubs collect the procedure arguments and results into packets for transmission across the network, which is otherwise invisible to the programmer, and unpack them at the other end. The shaded areas in Figure 2 are the same code that would be written by the programmer if the procedures and the calling program were to be linked together in a single program; the fact that they are actually on different machines is (almost) completely hidden by the stubs.

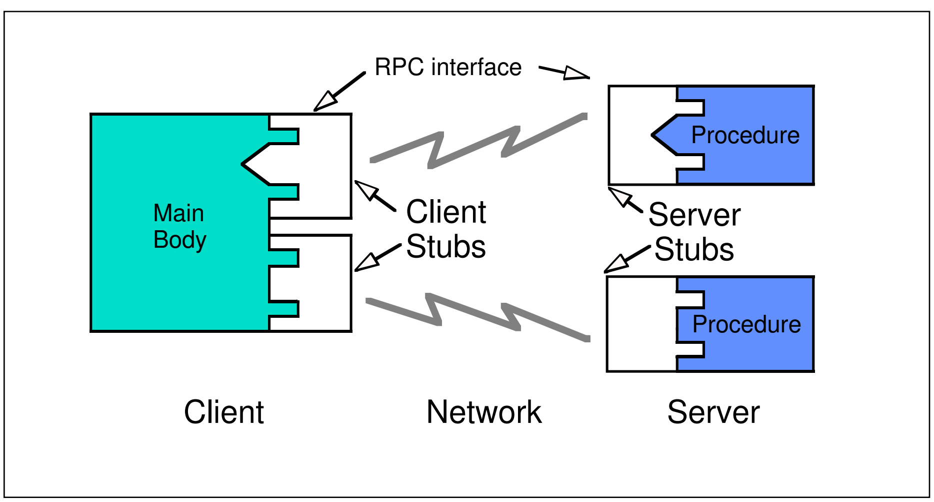

Figure 2. Remote Procedure Call

Individual procedures are grouped into interfaces. An IDL file describes an interface. This file is then compiled to produce source modules. When application programs are compiled, these source modules are included with the other source files written by the programmer.

3. Motivation

Hiding the underlying network simplifies programming but complicates design. The ease of making any procedure call a remote operation is a temptation to do so unwisely, and adds an entire new class of performance problems that do not succumb to ordinary techniques such as profiling. System administration is also more difficult. As Butler Lampson is rumored to have said: "A distributed system is one in which the failure of a machine I've never even heard of can cause my machine to fail." When a distributed system begins to perform badly, it is not at all obvious on what machine the problem actually lies. For this reason, instrumentation should be available in production code as well as during development and testing, so that system administrators can determine which component of a distributed system requires their attention. The difficulty of isolating the source and destination of RPCs makes inserting active instrumentation code into network interfaces worthwhile.

Inserting instrumentation code by hand causes some problems, however. Specifically, hand coding:

- Forces the programmer to write code specific to network calls, undermining the benefits that transparent RPC brought in the first place, particularly as local interfaces are changed to network interfaces and vice versa.
- Increases the potential for adding new bugs, as adding code always does.
- Makes some measurements, such as the time to actually marshal arguments, impossible to collect unless code is added to the stubs themselves; doing this by hand would be unacceptable in production code because the stubs could no longer be built from their (IDL) source.

These problems can be avoided, by having the IDL compiler automatically insert the necessary code.

4. Previous Work

Several earlier efforts have addressed this or related problems:

Blumson, et al. [3] modified the `rxgen` stub generator for the `rx` RPC protocol used by the AFS distributed file system to add code to automatically trace server calls. This work was done to generate traces of file server activity; no other instrumentation was added.

Hewlett-Packard's DCE Application Development Tools include an instrumented IDL compiler that adds tracing and logging code but does not accumulate operational statistics [4].

Madyam, et al., [5] describe a similar but less general extension to the DCE IDL compiler. They provide an extensive event tracing mechanism, but also do not accumulate statistics.

Friedrich and his coworkers [6] take a different approach to a similar goal. They place more general instrumentation code in carefully chosen locations in the DCE RPC run time library (RTL) that can capture and distinguish individual RPCs without adding code to each of the stubs. The choice between this approach and ours is a difficult one. Our approach adds code to each stub, but the RTL code is more complex and difficult to modify. Some measurements (e.g. marshalling time) are accessible to the stubs but not to the RTL while others (e.g. lengths of internal queues) are accessible to the RTL but not to the stubs. It was our belief that the IDL compiler changes were simpler (Friedrich et al. had access to the developers of the RTL) and more maintainable, but the latter is particularly subjective.

5. Data Collection Environment

The actual collection of data is done using IBM's Performance Toolkit for the RS/6000 [7]. The toolkit includes a number of components for transferring data from a number of machines being monitored and for creating graphical displays of the data being collected, but the component of interest to us is the System Performance Measurement Interface (SPMI). SPMI provides a set of functions that establish a connection between the instrumented process and a demon (`xmservd`) which collects data for the machine and manages communication with the outside world. The connection is buffered using shared memory.

Individual measurement points are named in a node-wide (actually network-wide) hierarchical name space; the actual structure of the hierarchy is not defined by the toolkit.

6. Design

To illustrate our instrumentation process, we use the `timop` interface example often used with DCE. The IDL for the `timop` interface is shown in Appendix A. It includes one procedure, `timop_getspan()`.

6.1 Changes to the Stub Procedures

Our IDL compiler has an additional command-line flag (named "SPMI" for obvious, if not very good, reasons) to insert instrumentation. Subflags indicate whether the client and/or server stubs should be instrumented, and select the data to be collected. Subflags can collect total elapsed time; marshalling and unmarshalling time; call counts by protocol sequence; and thread activity. Another subflag indicates whether the elapsed time instruments should collect sum of squares as well as totals.

Appendix B shows the server stub generated with full instrumentation; added sections are indicated by change bars in the left margin (the "`ifndef KERNEL`" directives around all of the insertions are to ensure compatibility with the DCE build processes, where the same stubs are sometimes used by both kernel and user processes; our collection mechanisms cannot be used inside the kernel).

The instrumentation itself is quite simple. For example, the lines:

```
#ifndef KERNEL
curtime(&unmarsh_now); /* Start server stub unmarshal time. */
#endif /* KERNEL */
rpc_ss_ndr_unmar_interp(
 2,/* number of [in] parameters */
 88,/* type vector index of first [in] parameter */
 IDL_param_vec, &IDL_ms);
```

```
#ifndef KERNEL
curtime(&now); /* Obtain current time. Stop server stub unmarshall time
*/
calcdiff_n(unmarsh_time, unmarsh_now, now); /* Calc server stub unmar-
shall time. */

#endif /* KERNEL */
```

in the procedure `op0_ssr()` (which implements `timop_getspan()`, operation zero in this interface), calculate the time required to unmarshal arguments. Similar snippets are added at obvious places. At the end of `op0_ssr` the various statistics are accumulated by the code:

```
if ( timops_mutex_exists && !pthread_mutex_lock(&timops_mutex) ) {

/* transfer our numbers to shared memory area for pickup */

if (s) {
s->op0_call_count++;
if (protocol_id == rpc_c_protocol_id_ncacn) {
s->op0_call_tcp++; /* count tcpip */
}
if (protocol_id == rpc_c_protocol_id_ncadg) {
s->op0_call_udp++; /* count udpdg */
}
s->op0_elapsed += call_time / 1000000000.0;
s->op0_elapsed_squared += call_time / 1000000000.0 * call_time /
1000000000.0;
s->op0_marsh += marsh_time / 1000000000.0;
s->op0_unmarsh += unmarsh_time / 1000000000.0;
s->op0_marsh_squared += marsh_time / 1000000000.0 * marsh_time /
1000000000.0;
s->op0_unmarsh_squared += unmarsh_time / 1000000000.0 * unmarsh_time /
1000000000.0;
s->op0_preemptions += preemption_stop - preemption_start;
s->op0_yields += yield_stop - yield_start;
s->op0_contexts += context_stop - context_start;
s->op0_utime += utime_time;
s->op0_stime += stime_time;
}
pthread_mutex_unlock(&timops_mutex);
```

The need for the mutual exclusion lock around this code was an object of some debate during our design. The values are actually collected by an asynchronous thread (described in more detail below) and the desire was to ensure that the statistics were always consistent so that, for example, dividing total elapsed time by the number of calls would always generate a correct average. It is not clear that the “off by one” errors that would occur without locking are actually a serious enough problem to justify the expense (and possible blocking) caused by the lock. On the other hand, on the server side we are able to delay this until after the response has already been returned to the caller.

6.2 Interface to SPMI

SPMI views the instrumented program as a single entity whereas the generated instrumentation knows only about the interfaces in a single IDL file. (A program may export many interfaces that are described in many IDL files.) To manage this, the compiler creates a procedure of predictable name (`IF_timops_inst_init()` in this example) which is known to the programmer. A list of these initialization procedures is passed to a general initialization procedure (`IF_init()`) to initialize the instrumentation system. This is the only hand generated code required by our instrumentation system. `IF_timops_inst_start()` and `IF_timops_inst_stop()` are used in a similar way to start and stop data collection.

SPMI requires that a shared memory buffer, known to both the instrumented process and to a collection demon, be updated periodically. A timing thread created by `IF_init()` performs this function. In order to support this, the addresses of the data accumulation structures and the mutual exclusion locks that guard them are returned to `IF_init` by the individual init routines and retained for use by the collection thread.

SPMI uses a hierarchical naming scheme. Individual measurements are described by structures such as:

```
{“callcnt0”, “Call Count”, 0, 1, SiCounter, SiULong, 1, SZ_OFF(timops_-
dat, op0_call_count, SiULong), NULL}
```

This example provides both a short and a displayable name, and indicates that this is a long integer counter. An array of these structures is constructed by the compiler, `timops_stats`. `IF_timops_inst_init()` then provides the name and description of the interface, which comprises the next level up in our naming hierarchy. `IF_init()` collects these and constructs the process context (the highest level of the process' naming subtree) name for the data, which will look something like:

```
{“DDS/IBM/<program name>.<process id>”, “Timops data”, 22, 0, timop_-
stats, STAT_L(timop_stats), NULL, 0, NULL, SiNoInst}
```

and registers it with SPMI. The process name is constructed from the program name and the process id, since there may be more than one copy of a program running on a machine, particularly for clients.

7. Results

Using our modified IDL compiler, we have been able to build a completely instrumented DCE cell. The compiler generates an enormous amount of data, some 4,000 different measurements could be produced by a fully instrumented but otherwise minimal DCE cell. Despite the amount of data generated, the impact of the inserted measurements is small. Using a `timop` server and client modified to have deterministic operating times, and after filtering some outlying values that are present whether instrumentation is included or not, we found the results shown in Table 1. The mean and standard deviation of the complete call time is shown with no instrumentation, with only the client instrumented, with only the server instrumented and with both instrumented. Further, each of these conditions is shown both with the client and server on the same machine and on different machines.

TABLE 1. Effects of Adding Instrumentation

Instrumented	none	none	client only	client only	server only	server only	both	both
Client/Server Machine	same	different	same	different	same	different	same	different
Mean	.0101	.0105	.0102	.0106	.0102	.0105	.0102	.0111
Standard Dev.	.0012	0012	0012	.0013	.0033	.0013	.0012	.0049

The effects of adding instrumentation are less than the experimental variation in our measurements, despite our (largely successful) efforts to control variation. The degradation caused by inserting instrumentation is insignificant.

We have been less successful at making the data useful; the number of different values is simply too large to comprehend using the sorts of graphic monitoring tools that are currently in vogue. For the most part, we have been reduced to simply logging data for a period of time and then using ad hoc tools to construct reports like the excerpt in Table 2, which shows the operation name, process, elapsed time, marshalling time, and unmarshalling time for a sampling of the most invoked operations on one of our server machines during one of our tests. Even with these limited tools, the instrumentation has proven useful; the occasional very slow RPCs alluded to earlier succumbed, in part, to visual inspection of these reports.

TABLE 2. Sample Data

Operation	Process	Elapsed Time	Marshalling Time	Unmarshalling Time
rpc__mgmt_is_server_listening	secd.27883	0.01042121	0.00184023	0.00074579
rpc__mgmt_is_server_listening	rpcd.8929	0.51947830	0.00052571	0.00281252
rpc__mgmt_is_server_listening	secd.27883	0.01007829	0.00180822	0.00075052
rpc__mgmt_is_server_listening	rpcd.8929	0.49718380	0.00049915	0.00311676
rpc__mgmt_is_server_listening	rpcd.8929	0.82199130	0.00050282	0.00267284
rpc__mgmt_is_server_listening	rpcd.8929	0.86198700	0.00048999	0.00255799
rpc__mgmt_is_server_listening	rpcd.8929	0.90410750	0.00046702	0.00246570
rpc__mgmt_is_server_listening	secd.27883	0.00769023	0.00139167	0.00056915

8. Conclusion and Directions for Further Work

We believe that we have found a way to create a useful and complete measurement system for DCE via a relatively simple path. We see no reason why this approach would not be equally useful for other programming systems that use automatically generated interfaces. The difficulty we see is in comprehending and interpreting the vast amounts of information that a large distributed system instrumented at this fine level of detail can produce.

This work enabled us to produce mountains of data with relatively little effort. Now we have to address the hard problem in distributed measurement systems: building tools that can help us comprehend the interactions that we now have the data to see.

References

1. Rosenberry, Ward, Kenney, David and Fisher, Gerry, *Understanding DCE,* O'Reilly & Associates. 1992.
2. Birrell, A. D. and Nelson, B. J., "Implementing remote procedure calls", in *Innovations in Internetworking*, Dedham, MA, Artech House, 1988.
3. Blumson, Sarr, Honeyman, Peter, Ragland, Thomas and Stolarchuk, Michael, *AFS Server Logging*. Ann Arbor, MI, CITI Technical Report 93-10.
4. *DCE Product Support Catalog*, Cambridge, MA, Open Software Foundation, 1994.
5. Mandyam. Sriram, Wei, Yi-Hsiu and Lin, David D. H.. "A Statistics Gatherizing and Analyzing for OSF's DCE". *IndustryProceedings: International Conference on DCE*, University of Karlsruhe, 1993.
6. Friedrich, Richard, Martinka, Joseph, Sienknecht, Tracy, Chelliah, Muthuswamy and Ranganathan, Aravindan, *A Distributed Performance Measurement System for Large-Scale Heterogeneous Environments*, Cupertino, CA, Hewlett-Packard Technical Report NSA-93-031.
7. *AIX Performance Toolkit/6000 User's Guide*, IBM Document Number SC23-2579-00, 1993.

Acknowledgements

This work was supported by a research contract from IBM. The advice and support of Dr. David Bachmann of IBM was invaluable. Many of our colleagues at CITI provided comments and suggestions, particularly Charles Antonelli. Mary Jane Northrop did her usual stellar editing.

Appendix A: IDL for the `timop` Interface

```
/* @(#)91 1.1 src/examples/timop/timop.idl, examples.src, aixdce12,
9324324 9/22/92 14:16:15 */
/*
 * COMPONENT_NAME: examples.src
 *
 * FUNCTIONS:
 *
 * ORIGINS: 72
 *
 *
 */
/*
** (c) Copyright 1990, 1991 OPEN SOFTWARE FOUNDATION, INC.
** ALL RIGHTS RESERVED
*/
/*
 * Permission is hereby granted to use, copy, modify and freely distrib-
ute
 * the software in this file and its documentation for any purpose with-
out
 * fee, provided that the above copyright notice appears in all copies
and
 * that both the copyright notice and this permission notice appear in
 * supporting documentation. Further, provided that the name of Open
 * Software Foundation, Inc. ("OSF") not be used in advertising or
 * publicity pertaining to distribution of the software without prior
 * written permission from OSF. OSF makes no representations about the
 * suitability of this software for any purpose. It is provided "as is"
 * without express or implied warranty.
 */

/*
**timop.idl
**
**IDL interface specification for remote time operations.
*/

/* We need explicit handles in timop because our client has multiple
(actually,
 multi-threaded) RPCs bound to multiple explicitly-specified servers. */

[uuid(0cf616d8-b858-11c9-8078-02608c0a03a7),
 version(1.0)]
interface timop
{
/* DTS timestamps are already in a universal format,
 so are opaque to (the presentation layer of) the RPC
 (16 = sizeof(utc_t)). */
```

```
const smallSIZEOF_TIMESTAMP = 16;
typedef bytetimestamp_t[SIZEOF_TIMESTAMP];

/* Failure value for remote status indications. */
const longTIMOP_ERR = -1;

/* Get the time span to do a job (random factorial). */
[idempotent]
void timop_getspan(
[in]handle_thandle,
[in]longrand,
[out]timestamp_ttimestamp,
[out]long*status_p,
[in,out] error_status_t*remote_status_p);
}
```

Appendix B: Sever Stub Generated with Full Instrumentation

The follwing code shows the server stub generated with full instrumentation; added sections are indicated by change bars in the left margin (the "`ifndef KERNEL`" directives around all of the insertions are to ensure compatibility with the DCE build processes, where the same stubs are sometimes used by both kernel and user processes; our collection mechanisms cannot be used inside the kernel).

```
File timop_s_spmi.h:

/* Generated by UMICH IDL compiler version OSF DCE T1.1.0-03 */

/* You must include a call to IF_inst_init(..., IF_timops_inst_init),
 a call to IF_inst_start(..., IF_timops_inst_start), and
 a call to IF_inst_stop(..., IF_timops_inst_stop)
 in your server application code. */


#include <sys/Spmidef.h>

#define SPMI_ERR_MUTEX_INIT_FAILED -1
#define SPMI_ERR_ADDCX_FAILED -2
#define SPMI_ERR_DELCX_FAILED -3

/* macro to calculate difference between t1 and t2 as nanoseconds */
#define calcdiff_n(doubletime, t1, t2) { \
long seconds; \
doubletime = (t2).tv_nsec - (t1).tv_nsec + \
((seconds=(t2).tv_sec-(t1).tv_sec)==0 ? 0 : 1000000000.0 * seconds); \
}

/* macro to calculate difference between t1 and t2 as microseconds */
#define calcdiff_u(doubletime, t1, t2) { \
long seconds; \
doubletime = (t2).tv_usec - (t1).tv_usec + \
((seconds=(t2).tv_sec-(t1).tv_sec)==0 ? 0 : 1000000.0 * seconds); \
}


/* SPMI variables and structures for timops server */

/* mutex, flag, and spmi data area */
static pthread_mutex_t timops_mutex;
static boolean timops_mutex_exists = FALSE;

static struct timops_dat {
long op0_call_count;
long op0_call_udp;
long op0_call_tcp;
float op0_elapsed;
```

```
float op0_elapsed_squared;
float op0_marsh;
float op0_marsh_squared;
float op0_unmarsh;
float op0_unmarsh_squared;
long op0_preemptions;
long op0_yields;
long op0_contexts;
float op0_utime;
float op0_stime;
} *s = NULL; /* this will point into the shared data area */


static const struct SpmiRawStat timops_stats[] = {
{“callcnt0”, “Call Count”, 0, 1, SiCounter, SiULong, 1,
SZ_OFF(timops_dat, op0_call_count, SiULong), NULL},
{“udpcallcnt0”, “UDP Call Count”, 0, 1, SiCounter, SiULong, 2,
SZ_OFF(timops_dat, op0_call_udp, SiULong), NULL},
{“tcpcallcnt0”, “TCP Call Count”, 0, 1, SiCounter, SiULong, 3,
SZ_OFF(timops_dat, op0_call_tcp, SiULong), NULL},
{“elapsed0”, “Total Time”, 0.0, 1, SiCounter, SiFloat, 4,
SZ_OFF(timops_dat, op0_elapsed, SiFloat), NULL},
{“elapsedsq0”, “Sum of Squares of Total Time”, 0.0, 1, SiCounter, Si-
Float, 5,
SZ_OFF(timops_dat, op0_elapsed_squared, SiFloat), NULL},
{“marshall0”, “Marshalling Time”, 0.0, 1, SiCounter, SiFloat, 6,
SZ_OFF(timops_dat, op0_marsh, SiFloat), NULL},
{“marshallsq0”, “Sum of Squares of Marshall Time”, 0.0, 1, SiCounter,
SiFloat, 7,
SZ_OFF(timops_dat, op0_marsh_squared, SiFloat), NULL},
{“unmarshall0”, “Unmarshalling Time”, 0.0, 1, SiCounter, SiFloat, 8,
SZ_OFF(timops_dat, op0_unmarsh, SiFloat), NULL},
{“unmarshallsq0”, “Sum of Squares of Unmarshall Time”, 0.0, 1, Si-
Counter, SiFloat, 9,
SZ_OFF(timops_dat, op0_unmarsh_squared, SiFloat), NULL},
{“preemptions0”, “Thread Preemptions”, 0, 1, SiCounter, SiULong, 10,
SZ_OFF(timops_dat, op0_preemptions, SiULong), NULL},
{“yields0”, “Thread Yields”, 0, 1, SiCounter, SiULong, 11,
SZ_OFF(timops_dat, op0_yields, SiULong), NULL},
{“contexts0”, “Thread Context Switches”, 0, 1, SiCounter, SiULong, 12,
SZ_OFF(timops_dat, op0_contexts, SiULong), NULL},
{“utime0”, “User CPU Time”, 0, 1, SiCounter, SiFloat, 13,
SZ_OFF(timops_dat, op0_utime, SiFloat), NULL},
{“stime0”, “System CPU Time”, 0, 1, SiCounter, SiFloat, 14,
SZ_OFF(timops_dat, op0_stime, SiFloat), NULL}
};

/* included from <dce_build_dir>/src/rpc/runtime/rpclist.h
 and <dce_build_dir>/src/rpc/runtime/com.h */
#define rpc_c_protocol_id_ncacn 0
#define rpc_c_protocol_id_ncadg 1
```

```
typedef struct
{
 void *linknext, *linklast;
 unsigned32 protocol_id;
} *rpc_binding_rep_p_t;
```

File timop_sstub.c

```
/* Generated by UMICH IDL compiler version OSF DCE T1.1.0-03 */
#ifdef VMS
#pragma nostandard
#endif
#include <timop.h>
#include <dce/idlddefs.h>
static idl_ulong_int IDL_offset_vec[] = {
0,/* sentinel */
0/* sentinel */
};

static void (*IDL_rtn_vec[])() = {
(void (*)())NULL,/* sentinel */
(void (*)())NULL/* sentinel */
};

static idl_byte IDL_type_vec[] = {
/* 0 */ 0xff,0xff,0xff,0xff,
/* 4 */ 1,/* little endian */
/* 5 */ 0,/* ASCII */
/* 6 */ 0xff,0xff,
/* 8 */ 0x03,0x00,/* interpreter encoding major version 3 */
/* 10 */ 0x01,0x00,/* interpreter encoding minor version 1 */
/* 12 */ 0x01,0x00,/* interface timop major version 1 */
/* 14 */ 0x00,0x00,/* interface timop minor version 0 */
/* 16 */ 0xd8,0x16,0xf6,0x0c,/* uuid time_low */
/* 20 */ 0x58,0xb8,/* uuid time_mid */
/* 22 */ 0xc9,0x11,/* uuid time_hi_and_version */
/* 24 */ 0x80,/* uuid clock_seq_hi_and_reserved */
/* 25 */ 0x78,/* uuid clock_seq_low */
/* 26 */ 0x02,0x60,0x8c,0x0a,0x03,0xa7,/* uuid node */
/* 32 */ 0x00,0x00,0x00,0x00,/* no storage information */
/* 36 */ 0x04,0x00,0x00,0x00,/* number of bug flags = 4 */
/* 40 */ 0x98,0x00,0x00,0x00,/* index of bug flags = 152 */
/* 44 */ 0xff,0xff,0xff,0xff,
/* 48 */ 0xff,0xff,0xff,0xff,
/* 52 */ 0xff,0xff,0xff,0xff,
/* 56 */ 0xff,0xff,0xff,0xff,
/* 60 */ 0x01,0x00,0x00,0x00,/* number of operations = 1 */
/* 64 */ 0x02,0x00,0x00,0x00,/* operation timop_getspan flags */
/* 68 */ 0x06,0x00,0x00,0x00,/* number of timop_getspan params = 6 */
/* 72 */ 0x02,0x00,0x00,0x00,/* number of timop_getspan [in]s = 2 */
/* 76 */ 0x58,0x00,0x00,0x00,/* index of timop_getspan [in]s = 88 */
```

```
/* 80 */ 0x03,0x00,0x00,0x00,/* number of timop_getspan [out]s = 3 */
/* 84 */ 0x68,0x00,0x00,0x00,/* index of timop_getspan [out]s = 104 */
/* Operation timop_getspan parameter rand */
/* 88 */ 0x02,0x00,0x00,0x00,/* long 2 index of parameter rand */
/* 92 */ IDL_DT_LONG,/* rand */
/* 93 */ IDL_DT_EOL,
/* Operation timop_getspan parameter remote_status_p */
/* 94 */ 0xff,0xff,/* filler */
/* 96 */ 0x05,0x00,0x00,0x00,/* long 5 index of parameter remote_sta-
tus_p */
/* 100 */ IDL_DT_PASSED_BY_REF,
/* 101 */ IDL_DT_ERROR_STATUS,/* error_status_t remote_status_p */
/* 102 */ IDL_DT_EOL,
/* Operation timop_getspan parameter timestamp */
/* 103 */ 0xff,/* filler */
/* 104 */ 0x03,0x00,0x00,0x00,/* long 3 index of parameter timestamp */
/* 108 */ IDL_DT_PASSED_BY_REF,
/* 109 */ IDL_DT_FIXED_ARRAY,
/* 110 */ 0,/* properties */
/* 111 */ 0xff,/* filler */
/* 112 */ 0x8b,0x00,0x00,0x00,/* long 139 full array index */
/* 116 */ 0x8b,0x00,0x00,0x00,/* long 139 flat array index */
/* 120 */ IDL_DT_EOL,
/* Operation timop_getspan parameter status_p */
/* 121 */ 0xff,0xff,0xff,/* filler */
/* 124 */ 0x04,0x00,0x00,0x00,/* long 4 index of parameter status_p */
/* 128 */ IDL_DT_PASSED_BY_REF,
/* 129 */ IDL_DT_LONG,/* status_p */
/* 130 */ IDL_DT_EOL,
/* Operation timop_getspan parameter remote_status_p */
/* 131 */ 0xff,/* filler */
/* 132 */ 0x05,0x00,0x00,0x00,/* long 5 index of parameter remote_sta-
tus_p */
/* 136 */ IDL_DT_PASSED_BY_REF,
/* 137 */ IDL_DT_ERROR_STATUS,/* error_status_t remote_status_p */
/* 138 */ IDL_DT_EOL,
/* array timestamp_t full instance */
/* 139 */ 1,/* num dimensions */
/* 140 */ 0x00,0x00,0x00,0x00,/* long 0 fixed bound */
/* 144 */ 0x0f,0x00,0x00,0x00,/* long 15 fixed bound */
/* 148 */ IDL_DT_BYTE,
/* 149 */ 0xff,0xff,0xff,/* filler */
/* 152 */ 0x00,0x00,0x00,0x10,/* bug flags */
0/* sentinel */
};

#ifndef KERNEL
#include <stdio.h>
#include <timop_s_spmi.h>

extern char SpmiErrmsg[];
```

```
extern int SpmiErrno;
#endif /* KERNEL */

static void op0_ssr
#ifdef IDL_PROTOTYPES
(
 handle_t handle,
 rpc_call_handle_t IDL_call_h,
 rpc_iovector_elt_p_t IDL_elt_p,
 ndr_format_p_t IDL_drep_p,
 rpc_transfer_syntax_p_t IDL_transfer_syntax_p,
 rpc_mgr_epv_t IDL_mgr_epv,
 error_status_t *IDL_status_p
)
#else
(handle, IDL_call_h, IDL_elt_p, IDL_drep_p, IDL_transfer_syntax_p,
IDL_mgr_epv, IDL_status_p)
 handle_t handle;
 rpc_call_handle_t IDL_call_h;
 rpc_iovector_elt_p_t IDL_elt_p;
 ndr_format_p_t IDL_drep_p;
 rpc_transfer_syntax_p_t IDL_transfer_syntax_p;
 rpc_mgr_epv_t IDL_mgr_epv;
 error_status_t *IDL_status_p;
#endif
{
IDL_ms_t IDL_ms;
volatile ndr_boolean IDL_manager_entered = ndr_false;
volatile RPC_SS_THREADS_CANCEL_STATE_T IDL_general_cancel_state;
idl_byte IDL_stack_packet[IDL_STACK_PACKET_SIZE];
rpc_void_p_t IDL_param_vec[6];
idl_long_int rand;
timestamp_t timestamp;
idl_long_int status_p;
error_status_t remote_status_p;
#ifndef KERNEL

/* SPMI variables for this remote procedure. */

/* Real Time Clock timers */
struct timestruc_t now, call_now, marsh_now, unmarsh_now;

/* nanosecond and microsecond elapsed times */
double call_time, marsh_time, unmarsh_time, /* nano */
 utime_time, stime_time; /* micro */

/* user and system cpu times */
struct timeval utime_start, utime_stop, stime_start, stime_stop;

/* counters */
long preemption_start, preemption_stop, yield_start, yield_stop,
```

```
 context_start, context_stop;

/* protocol id */
unsigned32 protocol_id;

#endif /* KERNEL */
#ifndef KERNEL
curtime(&call_now); /* Start server stub call time. */
#endif /* KERNEL */
#ifndef KERNEL
cma_thread_get_stats(&utime_start, &stime_start, &preemption_start,
&yield_start,
 &context_start); /* Get thread stats start count. */
#endif /* KERNEL */
RPC_SS_INIT_SERVER
rpc_ss_init_marsh_state(IDL_type_vec, &IDL_ms);
IDL_ms.IDL_stack_packet_status = IDL_stack_packet_unused_k;
IDL_ms.IDL_stack_packet_addr = IDL_stack_packet;
TRY
IDL_ms.IDL_offset_vec = IDL_offset_vec;
IDL_ms.IDL_rtn_vec = IDL_rtn_vec;
IDL_ms.IDL_call_h = (volatile rpc_call_handle_t)IDL_call_h;
IDL_ms.IDL_drep = *IDL_drep_p;
IDL_ms.IDL_elt_p = IDL_elt_p;
IDL_param_vec[1] = (rpc_void_p_t)&handle;
IDL_param_vec[2] = (rpc_void_p_t)&rand;
IDL_param_vec[3] = (rpc_void_p_t)timestamp;
IDL_param_vec[4] = (rpc_void_p_t)&status_p;
IDL_param_vec[5] = (rpc_void_p_t)&remote_status_p;
IDL_ms.IDL_param_vec = IDL_param_vec;
IDL_ms.IDL_side = IDL_server_side_k;
IDL_ms.IDL_language = IDL_lang_c_k;
#ifndef KERNEL
curtime(&unmarsh_now); /* Start server stub unmarshall time. */
#endif /* KERNEL */
rpc_ss_ndr_unmar_interp(
 2,/* number of [in] parameters */
 88,/* type vector index of first [in] parameter */
 IDL_param_vec, &IDL_ms);
#ifndef KERNEL
curtime(&now); /* Obtain current time. Stop server stub unmarshall time.
*/
calcdiff_n(unmarsh_time, unmarsh_now, now); /* Calc server stub unmar-
shall time. */
#endif /* KERNEL */

/* manager call */
IDL_manager_entered = ndr_true;
RPC_SS_THREADS_ENABLE_GENERAL(IDL_general_cancel_state);

 (*((timop_v1_0_epv_t *)IDL_mgr_epv)->timop_getspan)(
```

```
 handle,
 rand,
 timestamp,
 &status_p,
 &remote_status_p);
#ifndef KERNEL
/* Obtain the server side protseq (quick method) */
protocol_id = ((rpc_binding_rep_p_t)handle)->protocol_id;
#endif /* KERNEL */
RPC_SS_THREADS_RESTORE_GENERAL(IDL_general_cancel_state);
#ifndef KERNEL
curtime(&marsh_now); /* Start server stub marshall time. */
#endif /* KERNEL */
{
rpc_ss_ndr_marsh_interp(
 3,/* number of [out] parameters */
 104,/* type vector index of first [out] parameter */
 IDL_param_vec, &IDL_ms);
#ifndef KERNEL
curtime(&now); /* Obtain current time. Stop server stub marshall time.
*/
calcdiff_n(marsh_time, marsh_now, now); /* Calc server stub marshall
time. */
#endif /* KERNEL */
#ifndef KERNEL
calcdiff_n(call_time, call_now, now); /* Calc server stub call time. */
#endif /* KERNEL */
if (IDL_ms.IDL_iovec.num_elt != 0)
 rpc_call_transmit((rpc_call_han-
dle_t)IDL_ms.IDL_call_h,(rpc_iovector_p_t)&IDL_ms.IDL_iovec,
 (unsigned32*)&IDL_ms.IDL_status); /* Send remaining outs */

}
IDL_closedown: ;
CATCH_ALL
rpc_ss_ndr_clean_up(&IDL_ms);
if (!RPC_SS_EXC_MATCHES(THIS_CATCH,&rpc_x_ss_pipe_comm_error))
{
if ( ! IDL_manager_entered )
{
}
{
rpc_ss_send_server_exception_2(IDL_call_h,THIS_CATCH,0,NULL,&IDL_ms);
IDL_ms.IDL_status = error_status_ok;
}
}
ENDTRY
if (IDL_ms.IDL_mem_handle.memory)
{
 rpc_ss_mem_free(&IDL_ms.IDL_mem_handle);
}
```

```
if (IDL_ms.IDL_status != error_status_ok)
{
if (IDL_ms.IDL_status == rpc_s_call_cancelled)
{
rpc_ss_send_server_exception(IDL_call_h,&RPC_SS_THREADS_X_CANCELLED);
IDL_ms.IDL_status = error_status_ok;
}
else
{
{
rpc_ss_send_server_exception(IDL_call_h,&rpc_x_ss_remote_comm_failure);
IDL_ms.IDL_status = error_status_ok;
}
}
}
*IDL_status_p = IDL_ms.IDL_status;
#ifndef KERNEL
cma_thread_get_stats(&utime_stop, &stime_stop, &preemption_stop,
&yield_stop,
 &context_stop); /* Get thread stats end count. */
calcdiff_u(utime_time, utime_start, utime_stop); /* calculate user cpu
time */
calcdiff_u(stime_time, stime_start, stime_stop); /* calculate system cpu
time */
#endif /* KERNEL */
#ifndef KERNEL

if ( timops_mutex_exists && !pthread_mutex_lock(&timops_mutex) ) {

/* transfer our numbers to shared memory area for pickup */

if (s) {
s->op0_call_count++;
if (protocol_id == rpc_c_protocol_id_ncacn) {
s->op0_call_tcp++; /* count tcpip */
}
if (protocol_id == rpc_c_protocol_id_ncadg) {
s->op0_call_udp++; /* count udpdg */
}
s->op0_elapsed += call_time / 1000000000.0;
s->op0_elapsed_squared += call_time / 1000000000.0 * call_time /
1000000000.0;
s->op0_marsh += marsh_time / 1000000000.0;
s->op0_unmarsh += unmarsh_time / 1000000000.0;
s->op0_marsh_squared += marsh_time / 1000000000.0 * marsh_time /
1000000000.0;
s->op0_unmarsh_squared += unmarsh_time / 1000000000.0 * unmarsh_time /
1000000000.0;
s->op0_preemptions += preemption_stop - preemption_start;
s->op0_yields += yield_stop - yield_start;
s->op0_contexts += context_stop - context_start;
```

```
s->op0_utime += utime_time;
s->op0_stime += stime_time;
}
pthread_mutex_unlock(&timops_mutex);

}
#endif /* KERNEL */
}

static rpc_v2_server_stub_proc_t IDL_epva[] =
{
 (rpc_v2_server_stub_proc_t)op0_ssr
};

static rpc_syntax_id_t IDL_transfer_syntaxes[1] = {
{
{0x8a885d04, 0x1ceb, 0x11c9, 0x9f, 0xe8, {0x8, 0x0, 0x2b, 0x10, 0x48,
0x60}},
2}};

static rpc_if_rep_t IDL_ifspec = {
 1, /* ifspec rep version */
 1, /* op count */
 1, /* if version */
 {0x0cf616d8, 0xb858, 0x11c9, 0x80, 0x78, {0x2, 0x60, 0x8c, 0xa, 0x3,
0xa7}},
 2, /* stub/rt if version */
 {0, NULL}, /* endpoint vector */
 {1, IDL_transfer_syntaxes} /* syntax vector */
,IDL_epva /* server_epv */
,NULL /* manager epv */
};
/* global */ rpc_if_handle_t timop_v1_0_s_ifspec = (rpc_if_handle_t)&ID-
L_ifspec;
#ifdef VMS
#pragma standard
#endif
#ifndef KERNEL

static int cx_index;
static char program[64];

int
IF_timops_inst_init
#ifdef PROTO
(
int i,
char *progname,
pthread_mutex_t **mtx,
struct timops_dat ***shp,
char *pname, *desc,
```

```
const struct SpmiRawStat (**ptr)[],
int *num
)
#else
(i, progname, mtx, shp, pname, desc, ptr, num)
int i;
char *progname;
pthread_mutex_t **mtx;
struct timops_dat ***shp;
char *pname, *desc;
const struct SpmiRawStat (**ptr)[];
int *num;
#endif
{
int pid = getpid();
if ( pthread_mutex_init(&timops_mutex, pthread_mutexattr_default) )
return SPMI_ERR_MUTEX_INIT_FAILED;
timops_mutex_exists = TRUE;
/* save input parameter */
cx_index = i - 1;
/* return output parameters */
*mtx = &timops_mutex;
*shp = &s;
sprintf(pname, "DDS/IBM/%s.%d/timops", progname, pid);
sprintf(program, "DDS/IBM/%s.%d/timops", progname, pid); /* save a local
copy */
sprintf(desc, "timop server data");
*ptr = &timops_stats;
*num = STAT_L(timops_stats);
return 0;
}

int
IF_timops_inst_start()
{
if ( timops_mutex_exists && !pthread_mutex_lock(&timops_mutex)) {
if (!(s = (struct timops_dat *)SpmiDdsAddCx(cx_index, program, "timops
interface", cx_index))) {
#ifdef DEBUG
fprintf(stderr, "SpmiDdsAddCx failed: %d %s", SpmiErrno, SpmiErrmsg);
#endif /* DEBUG */
pthread_mutex_unlock(&timops_mutex);
return SPMI_ERR_ADDCX_FAILED;
}
/* Init counters */
s->op0_call_count = 0;
s->op0_call_tcp = 0;
s->op0_call_udp = 0;
s->op0_elapsed = 0.0;
s->op0_elapsed_squared = 0.0;
s->op0_marsh = 0.0;
```

```
s->op0_marsh_squared = 0.0;
s->op0_unmarsh = 0.0;
s->op0_unmarsh_squared = 0.0;
s->op0_preemptions = 0;
s->op0_yields = 0;
s->op0_contexts = 0;
s->op0_utime = 0.0;
s->op0_stime = 0.0;

pthread_mutex_unlock(&timops_mutex);
}
return 0;
}

int
IF_timops_inst_stop()
{
if ( timops_mutex_exists && !pthread_mutex_lock(&timops_mutex) ) {
if (SpmiDdsDelCx((char *)s)) {
fprintf(stderr, “SpmiDdsDelCx failed: %d %s”, SpmiErrno, SpmiErrmsg);
pthread_mutex_unlock(&timops_mutex);
return SPMI_ERR_DELCX_FAILED;
}
s = NULL;
pthread_mutex_unlock(&timops_mutex);
}
return 0;
}
#endif /* KERNEL */
```