

BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-97-M15

“The Performance of Large Software Systems: A Case Study”
by
Peng Dai

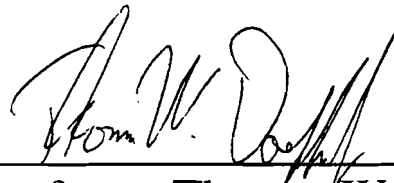
The Performance of Large Software Systems: A Case Study

Peng Dai

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in the Department of
Computer Science at Brown University

May 1997



Professor Thomas W. Doepfner Jr.

Advisor

1. Introduction

As the size of a software system increases, so do the complexity and frequency of interaction between its components and thus the chances of a mismatch from the performance perspective. Things are even worse when these components are from libraries that were developed at different times and for various purposes. It requires a lot of effort to optimize the overall performance of these components. Sometimes, when optimization is achieved under one condition, it fails miserably when the condition changes. Therefore, effective tools aiding in the understand of software systems, especially those large ones, are desirable.

The performance study of large software systems may be carried out at two different levels. Studies at the higher level imposes little requirement on the semantics of the target system. Analyses are conducted once the caller-callee relationship and the cost of various function calls are known. An example is the Paradyn parallel performance measurement tool developed by Miller et.al.^[11] The other approach is to make full use of the semantics of the system being studied and combine them with the function call data to provide semantically relevant results. The second approach requires intimate knowledge of the system, and is therefore considered at a lower level. Examples making use of this approach include Jakiela's distributed database study^[10] and various other case studies.

The benefit of the first approach is obvious. It has less dependence on the target systems, and therefore, has a wider application. However, the lack of relevant semantic knowledge sometimes makes the results coming out of this approach hard to understand. In other words, the results it produces may not necessarily answer the question programmers are interested in most - why the application is not as fast. The second approach exchanges generality for more relevant results. In this sense, the two approaches somehow complements each other.

In this paper, we focus on how to use the semantic knowledge we gained from our experience with the Distributed Computing Environment, or DCE, to diagnose performance anomalies encountered by various applications.

We start with the introduction of a couple of instrumentation techniques in section 2, which are used later in our study. These techniques include interpositioning and large-scale instrumentation.

In the section that follows, a brief introduction of DCE is provided. After that, we present both an architectural and a component-wise look at the system we have developed for performance analysis. We conclude this section with some interesting results we found by applying the tools to sample applications.

Finally, we summarize the work we have done so far and discuss the potential directions along which the current work might be extended.

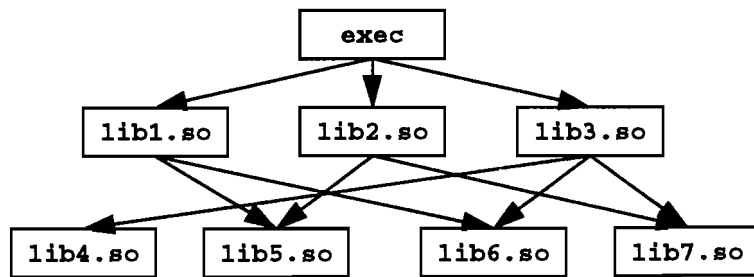


FIGURE 1. Shared Library Dependency Graph

The technique results in a breadth-first ordering of all dependencies, as shown below.

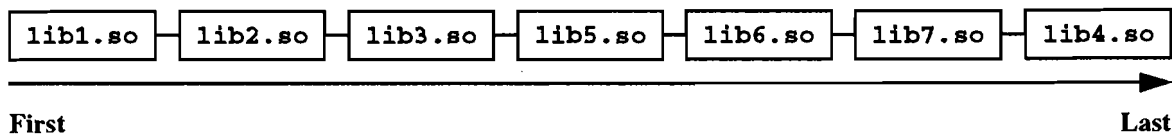


FIGURE 2. Shared Library Mapping

Symbol lookups are performed by searching in each object, starting with the dynamic executable, and progressing through each shared library in the same order in which they are mapped. As we will see later, it is this mechanism of searching for a symbol that enables interpositioning. Before that, let us first take a closer look at how symbols are processed by the runtime linker.

2.1.3. Symbol Resolution and Relocation Processing

When a function call is made or a variable is assigned to, the symbolic name of either the function or the variable has to be translated to the in-memory address in order for the operation to be realized. This process is generally referred to as relocation.

Consider the following example of relocation. In an object file, the section named `.foo` contains several references to symbol `bar`. For each reference, there is a corresponding record in a special section `.rela.foo` that describes how relocation is to be performed once the in-memory address of `bar` is known. The information in the record includes the location of the symbol reference, the type of the relocation to be performed, and an index in the symbol table pointing to the symbol being referenced. During the link-editing or runtime linking, depending on the nature of symbol and the containing object file, each relocation record is processed to update the symbolic reference to reflect the in-memory address of the symbol `bar`. Without relocation, the execution of instructions in an executable could not be properly carried out. The relocation process is described in the following figure.

2.1.4. Interpositioning

With the above background introduction on symbol processing, we are now ready to introduce the interpositioning technique.

As we said, the first time a global function defined in one of the shared library dependencies of the dynamic executable is called upon, the control is transferred to the runtime linker through the `.plt` entry, which then looks for the function definition in the chain of dependencies. If one definition of the function is found, the search is satisfied. Therefore, if more than one instances of the same function definition exists in multiple dependencies, the first instance will interpose on all other.

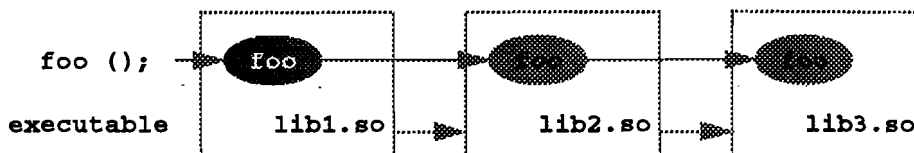


FIGURE 5. Interpositioning

This technique provides us almost unlimited capabilities to interpose existing function definition, thus either modifying or enhancing the original functionality, without even touching the shared library that contains it. We could achieve this by either relink the dynamic executable with interposing library being put before the interposed, or by using the command line interface discussed later.

One caveat, if any, is that this technique fails to deal with static or local functions whose referencing uses relative addressing.

2.1.5. Runtime Linking Command-line Interface

To successfully link relocatable object files and archive or shared libraries together to form an executable, the link-editor has to be informed of the path of the participating objects. In the case of archive or shared library, the path can be supplied with the `-L` command-line option. In the same vein, the runtime linker has to be aware of the location of the shared libraries that are recorded as dependencies during link-editing and have to be located again during process initialization and execution. Another command-line option `-R` is used just to record these so-called runpaths.

The order in which the shared library dependencies are recorded is significant. As we have seen above, the interpositioning technique depends heavily on it. Careful thought has to be exerted during link-editing to make sure the shared library dependencies are supplied in the right order. Commands that will help display the shared library dependencies in the order they were recorded include `ldd` and `dump`.

The runtime linker also allows additional objects to be introduced during process initialization. The environment variable `LD_PRELOAD` can be initialized to shared library or relocatable object file name, or a string of file names separated by white space. These objects will be mapped after the dynamic executable and before any shared library dependencies.

The functionality of the function `foo` in `libfoo.so` can be augmented by preloading a library `libnewfoo.so` which contains a new definition of function `foo` as follows.

```
#include <dlfcn.h>
#include <stdio.h>

double foo (int i)
{
    static double (*fptr) (int i) = 0;

    if (fptr == 0) {
        fptr = (double (*) (int))
            dlsym (RTLD_NEXT, "foo");

        if (fptr == NULL)
            abort ();
    }

    fprintf (stderr, "Input value = %d", i);

    return (*fptr) (i);
}
```

FIGURE 7. Function `foo` Definition

Finally, the above code fragment is placed into the shared library `libnewfoo.so`, which is then preloaded during the initialization of `bar`.

```
$ cc -o libnewfoo.so -G -K pic newfoo.c
$ LD_PRELOAD=./libnewfoo.so bar
```

FIGURE 8. Interposing Function `foo`

2.2. Large Scale Instrumentation

Needless to say, the new interpositioning technique makes instrumenting existing applications much easier without compromising the execution speed. To instrument the functions in one of the shared library dependencies of the dynamic executable, all we need to do is to generate, for each target function, an interposing unit, including a front end, which is an entry point exposed to the executable, an event generator, dispatcher, and processor, as shown in the following figure. The front end is a function with the same name as the back end, or the target function. The event generator may emit event record at various stages of the execution of the back end. Typically, these events represent the start, end, or exception status. The event dispatcher serves as a link between the front end and the event processor by examining the event and passing it on to the appropriate processor.

With the trace record defined above, we will be able to identify any function unambiguously. In case when two functions in different shared libraries share the same name, the most significant byte helps distinguish them. Even when two functions with the same name are defined in the same shared library, which is possible if at least one of them is static, we can make them out by assigning different facility codes to them.

In addition, the least significant op bit provides a simple characterization of the critical execution points, namely, start, end, and exception. It may not be as comprehensive as the traditional tracing technique can cover, which may place trace points virtually anywhere in the source code. But this is a limitation imposed by the interpositioning technique, not the trace record definition.

Last but not the least, the size of the trace record dramatically reduces the amount of bookkeeping and the overhead of processing. For example, the space required to maintain a function call stack using trace record can be roughly one order of magnitude less than using function names. And comparing stack elements is simply integer comparison instead of the more lengthy string comparison.

The model of computation DCE supplies is quite straightforward. A server decides which services it is willing to provide. Once the decision is made, an interface, which is basically a set of procedure declarations, has to be prescribed to describe how the services can be accessed. The interface serves as a contract between the client and the server. The client has to satisfy the terms of the contract in order to obtain help from the server. In DCE, an interface is written in a special language called the Interface Definition Language, or IDL. The compiler for the IDL generates from the interface a C header file, which contains the C language correspondent of the operation declarations, and special object files called stubs, which incorporate the machinery for communication to and from the remote party. The automated stub generation from the much simpler interface definition very much alleviates the programmer from the drudgery of laying down the frame of communication for each remote service.

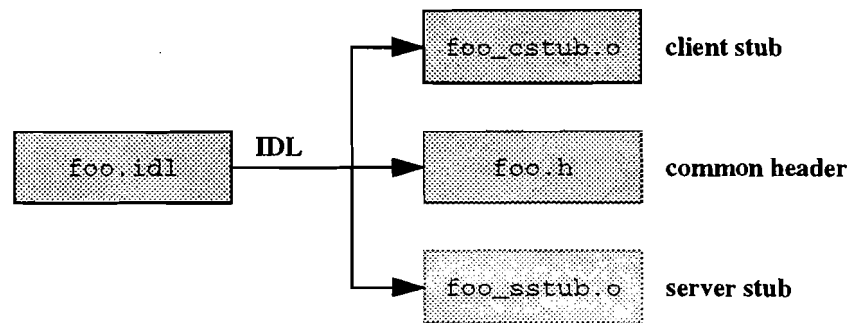


FIGURE 12. Interface Definition

After putting down the interface, the server may advertise the services by registering the interface with the directory service. A client, after locating the server providing the matching services, can invoke the services by making an RPC to the specific server. An RPC hides the details of network communication to simplify the application code. Such details include converting data between formats for different systems, converting application data structures to streams of bytes and back, detecting communication errors, possibly locating the right server instance and necessary security checks. Most of the details are handled inside the stub code which is automatically generated from an IDL file containing the interface description.

The following figure shows the layers of code an RPC goes through. The client application starts an RPC by making a local procedure call to the client side stub, which in turn communicates with the server side stub using the facilities provided by the runtime library. The server's RPC runtime library receives the RPC request and hand over the client information to the server stub which invokes the manager in the server application.

mission of input parameters and waits for the first packet containing the response to be delivered back by calling `rpc_call_transceive`. Following that, additional response data are unmarshaled in the call to `rpc_ss_ndr_unmar_interp`. Finally, the RPC is completely with a call to `rpc_call_end`. In case an exception is reported in any of these calls, it is caught and the RPC is finalized with a call to `rpc_call_end`.

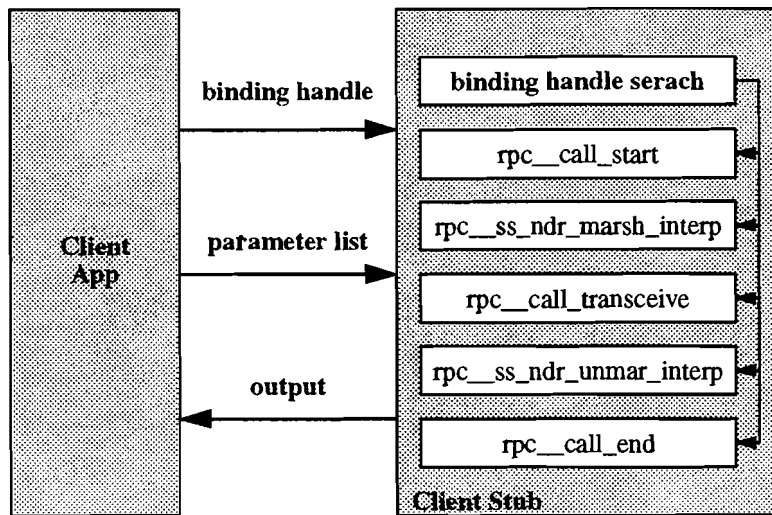


FIGURE 14. Inside the Client RPC

The protocol independent calls, such as `rpc_call_start`, `rpc_call_transceive`, and `rpc_call_end`, eventually calls the corresponding protocol dependent version, whose name contains the protocol tag. For example, in case of TCP, they are `rpc_cn_call_start`, `rpc_cn_call_transceive`, and `rpc_cn_call_end`; in case of UDP, `dg` is substituted for `cn` instead. The protocol information contained in the binding handle decides which is used. Although the names all look similar, the functionality can differ dramatically.

In case the binding handle returned from CDS or passed as a parameter is not fully bound, which means the end point is missing, a separate RPC has to be made to the RPC daemon running on the server machine to resolve it. Conditionally, if the client requests a secure RPC, it is time now to contact the security server to obtain the authentication ticket on behalf of the client. No matter what the underlying protocol is, the above actions have to be carried out if necessary.

The differences protocols may cause come next. If the RPC is using TCP as the message transport, an association between the client and the server has to be established before the call. An association is the application correspondent of a transport connection, which contains specific information that two communicating applications have agreed upon. In case of an RPC, the information may include the security context and the presentation syntax. Included in an association is a thread, which is named receiver for it is responsible for receiving and dispatching incoming packets. Negotiating an association could be potentially time consuming; this is why DCE has decided to optimize the process. An association and its underlying connection is established when one does not exist; but it preserves after the RPC that initiated it fin-

tor thread is notified and given the request. In the latter case, the listener thread, which listens for incoming connection request, reinitializes an unused association control block and wakes up its receiver, if one is available, or creates one from scratch. The receiver thread, when started, performs the same action as it is in the former case. After the call request, which is the first packet for a certain call, is handled, the receiver thread waits for further packets for the call and handles them accordingly. If the packet represents a cancel request and the executor thread has already started executing it, a cancellation will be delivered to it; otherwise, the original call request is dequeued and the receiver along with the association is freed. Normally, the executor thread detaches from the receiver thread and the association by calling `rpc_cn_call_end` or `rpc_dg_call_end`, from when on the receiver is free to accepting another RPC request. The server side flow of control in a typical connection-oriented RPC is shown in the following figure.

has an associated authentication identity that is fixed for its lifetime. In other words, each different authentication identity requires a unique activity ID.

The first time the client calls a server, a new client connection is created along with the activity ID. The activity ID is transmitted to the server which caches it for later reference. The server then calls back to the client to ask for its authentication identity, which is dubbed WAY for "Who Are You". After acquiring the authentication identity of the client, the server stores it with the previously obtained activity ID so that any subsequent call using the same activity ID does not incur a WAY authentication. In summary, servers identify clients by activity IDs and their associated authentication identity. Since only server cached activity IDs can prevent WAY authentication call-back, the client is encouraged to keep using existing activity IDs whenever possible. The sequence number in the client or server connection provides a means to implement non-idempotent calls. It is part of the purpose of WAY callback if unknown and is also cached by servers together with the activity ID and authentication identity.

Another important use of activity ID is when the underlying communication is shared among various ongoing RPCs. Its uniqueness allows the packets going through the shared channel to be dispatched properly.

In addition to the client and server connections, other data structures, such as client call handle and server call handle are used to provide both on-going RPC and inter-RPC cache of useful information. The relationship between various data structures used to implement a connectionless RPC is shown in the following figure.

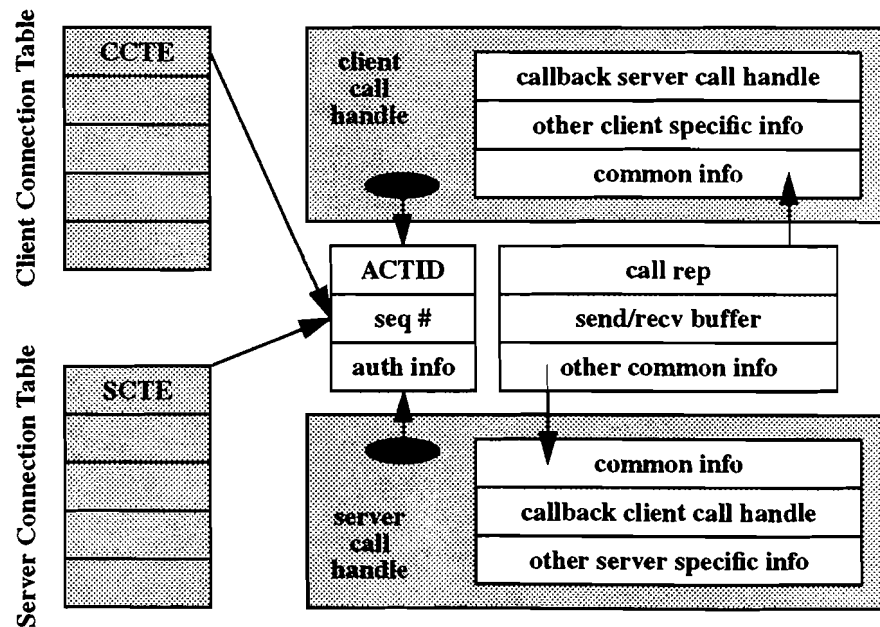


FIGURE 17. Data Structures for Connectionless RPC

An RPC using UDP as the underlying protocol starts with the allocation of a client call handle by calling `ccall_alloc` if one does not exist in the binding handle parameter or the existing one contains a different authentication identity or the socket included is disabled; otherwise, the existing one is reinitialized with updated per-call information, such as interface id and operation number. Inside `ccall_alloc`, a reference to a datagram socket is first obtained. Failure to obtain a socket leads to early

On the server side, the listener thread is waiting in `pthread_Select`. It is awoken by incoming packets on the datagram socket. It then uses the activity ID contained in the packet header to locate the corresponding server call handle. If this is the first time the client has ever used the same activity ID to contact the server, it will not be found in the cache. Therefore, a new server call handle will be created and the activity ID be cached. The listener then dispatches and handles the packet according to its content. Possible actions include `rpc__dg_do_request` for call request, `rpc__dg_do_quit` for quit request, `rpc__dg_do_ack` for acknowledgment request, `rpc__dg_do_ping` for ping request. In case of a call request, the call request is either handed over to an executor thread or enqueued on the call-waiting queue, depending on the availability of an idle executor thread. An executor thread starts executing by calling `rpc__dg_execute_call`. After interface lookup and object type inquiry are done, it checks whether the client authentication identity has been verified. If the activity ID and its associated authentication identity is cached, nothing needs to be done; otherwise, a WAY callback has to be performed before the stub is entered.

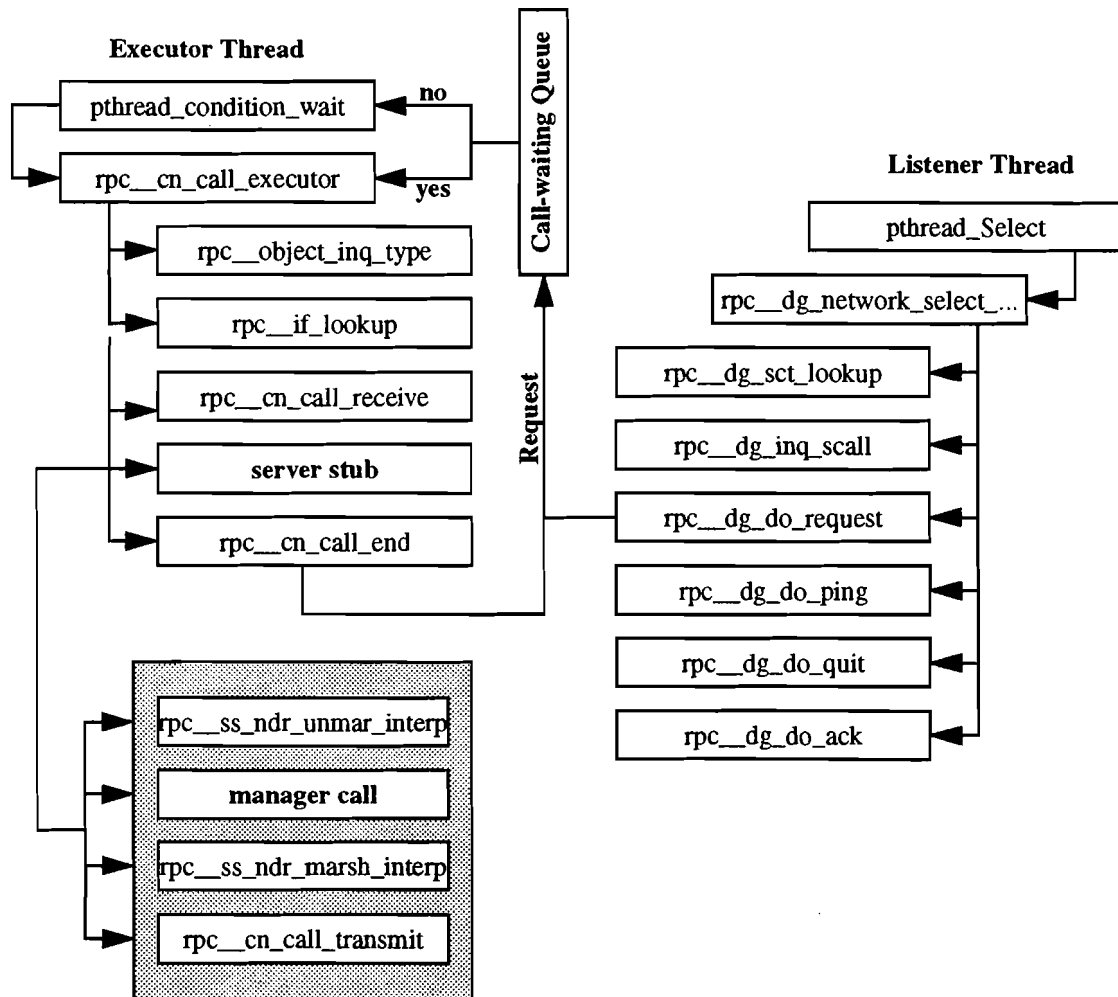
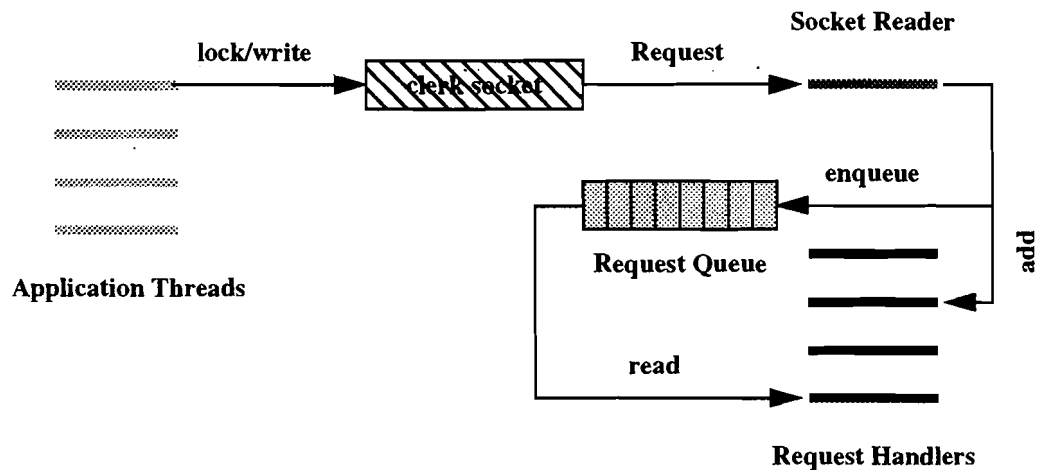


FIGURE 19. Server Side Connectionless RPC Implementation

FIGURE 20. Name Service Request Handling Mechanism

When multiple threads in the application make simultaneous requests to the name service, a lock has to be acquired to write to the cdsclerk socket. A complete request has to be written before the lock may be released for others to use. One thread in the cdsclerk is constantly reading the socket for incoming request. It also manages a pool of request handler threads. Before a request handler thread completes the current request, it checks to see if there are more in the request queue before it exits. If all request handler threads are busy when a request comes in, a new one will be created by the master reader thread. The interaction between the application and the cdsclerk is shown in the following figure.

**FIGURE 21. Application and Cdsclerk Interaction**

3.2.5. DCE Security

DCE security is based upon MIT Kerberos secret key approach. The runtime supports secured RPC by calling security server to acquire authentication information on behalf of the client when it is informed to. The server has to implement its own security policy which depends on the authentication information embedded in the call packet from the client.

3.3. System Overview

In the following, we will start from an architectural overview of the performance monitoring system and proceed with a detailed look at the assortment of components, including the functionality they provide, the techniques used during their construction, and the motivation behind the them.

3.3.1. Architecture

The architecture of an instrumented DCE application is shown in the following figure. The standard DCE facilities are provided indirectly to the application using object-oriented technology. The idea is that the application first constructs one or more objects representing a DCE service or utility, and then invoke the object member

Next, let us take a look at how an instrumented DCE application is to be linked and the shared libraries involved in the linking. Suppose the application serves requests over the interface `foo`. It will be linked against the following libraries in the order provided: `libifoo`, `libfoo_sstub`, `libtempl`, `libdcethread`, `libdwrpc`, `libdwexcep`, and `libinterp`, before the standard system libraries, where `libifoo` is automatically generated to interpose the server stubs, `libfoo_sstub` contains the server stubs, `libtempl` is a generic template library, `libdcethread` and `libdwrpc` provides object-oriented support, `libdwexcep` deals with DEC exceptions, and `libinterp` contains the main instrumentation. Although hidden from the programmer, `libinterp` is linked against `libmon`, which processes trace events and communicate with the display, and `libdwexcep`. The shared library component hierarchy is illustrated in the following figure.

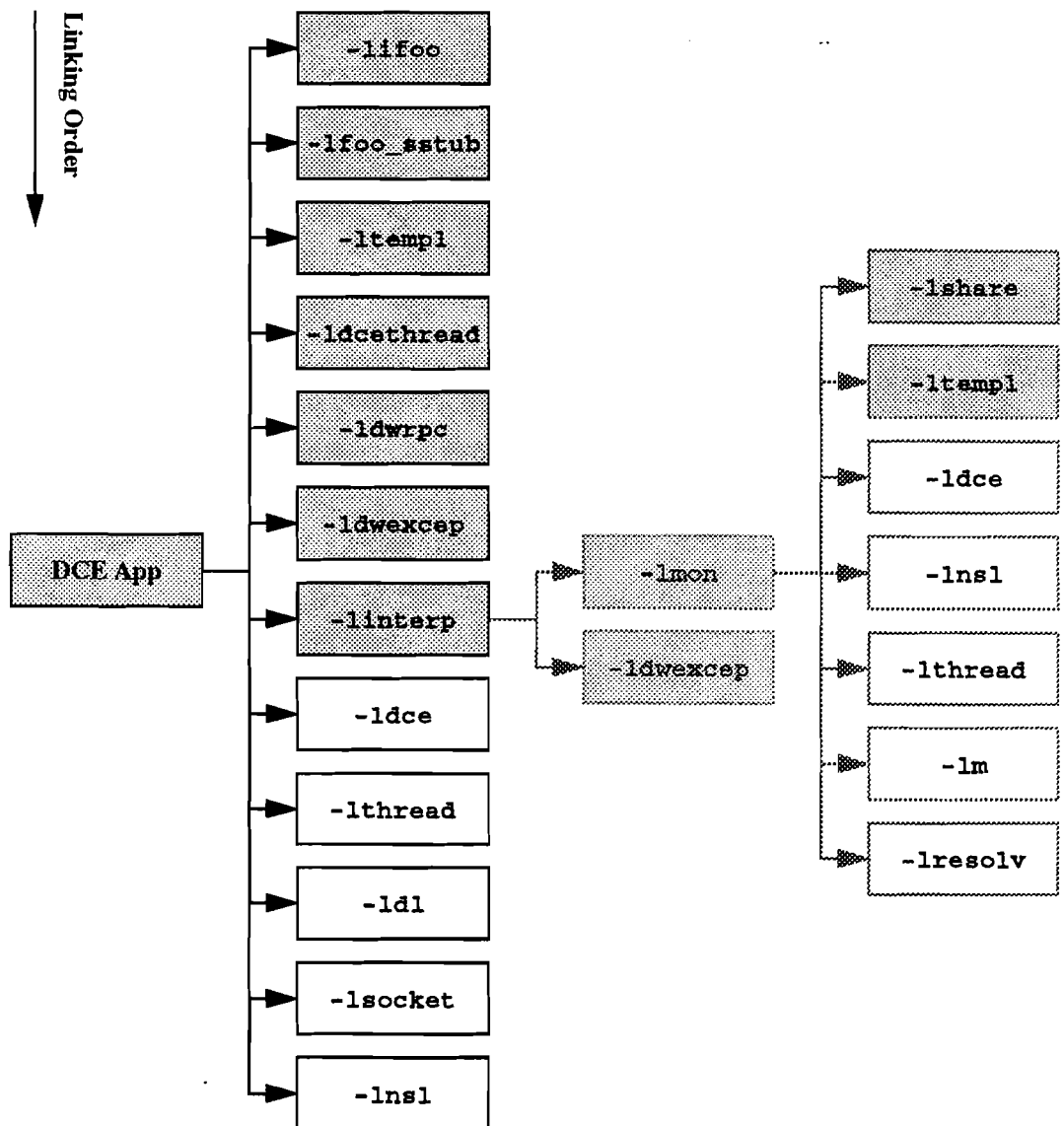


FIGURE 23. DCE Application Shared library Component Hierarchy

3.3.3. Interpositioning Library

The shared library `libinterp` provides the machinery to intercept function calls from the application to the DCE runtime. When a function call is intercepted for the first time, the next symbol in the dependency list is looked up and stored as a static variable for use in subsequent calls. The call to the original function is arranged after facilities are set up for the purpose of tracing. Currently, trace points are placed both before the call and after. Besides, exceptional conditions are monitored and exceptions thrown from the call are caught, recorded, and reraised.

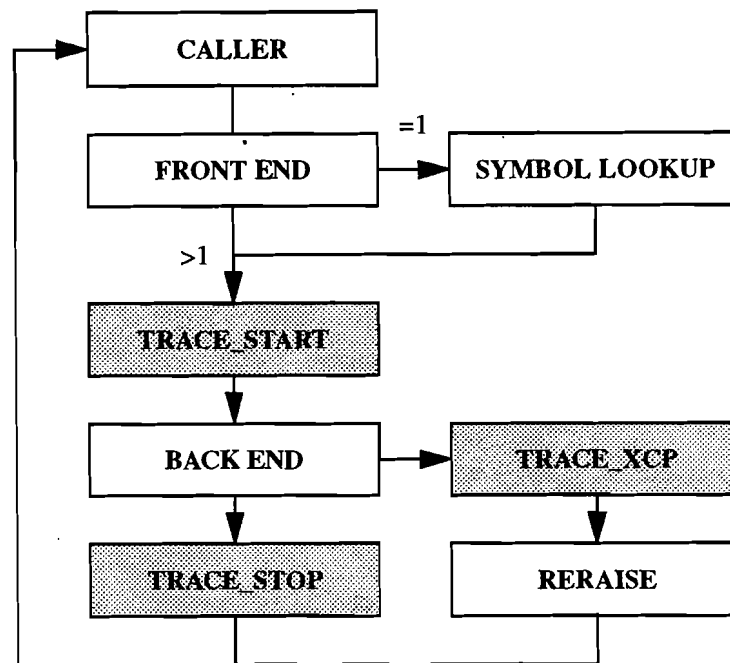


FIGURE 24. Interpositioning

The tracing facilities invoked from inside `libinterp` is provided by another shared library, `libmon`. An instrumented DCE application is not required to be linked against `libmon` explicitly. Instead, it is referenced by the application implicitly as part of the dependencies of `libinterp`.

The design of the tracing facilities, include trace event logging, processing, and displaying, is centered around the core object model, of which the concept of an observable and an observer is the corner stone. An observable, as the name suggests, describes any object that can be observed. A number of attributes are attached to an observable. In addition, an observable is given an identity called object key, which distinguishes itself from other observables, and an optional character name. The task of tracing, as prescribed in this model, is to control the creation and deletion of the observables, and computing the attribute values from the relevant trace events. An observer is the counterpart of an observable on the display. It includes the attribute values and the interface component and has the same object key as the observable it is observing. The correspondence between an observable and an observer is shown in the following figure. Examples of an observable include threads, sockets, processes, etc.

Appendix C: Grammar Used for Automatic IDL Conversion

```

%{
#ifdef IDLGRAMMAR_HEADER
#define IDLGRAMMAR_HEADER

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream.h>
#include <fstream.h>
#include "IDLlast.H"
#include "IDLInterp.H"

extern char *yytext;
int yyparse ();
int yylex ();
int param_count = 0;
%}

%token ID
%token STAR
%token COMMA
%token LPAREN
%token RPAREN
%token EXTERN
%token SEMICOLON
%token ARRAYBOUND

%union {
    unsigned short intval;
    char *string;
    class IDLlastIdNode *id;
    class IDLlastBoundNode *bound;
    class IDLlastTypeNode *tp;
    class IDLlastFuncTypeNode *ftp;
    class IDLlastParamNode *parameter;
    class IDLlastParamListNode *parameters;
    class IDLlastOpNode *op;
    class IDLlastOpListNode *ops;
}

%type <string> ID ARRAYBOUND
%type <intval> pointer type_pointer
%type <tp> type
%type <bound> bound array_bound
%type <id> type_id name param_name
%type <parameter> param
%type <parameters> param_list
%type <op> decl
%type <ops> decl_list

%start start
%{
#endif
%}

%%
start:
    decl_list
    {
        IDLlastNode::declare ();
        iidl_>header () << *$<ops>1 << endl;
    }

```



```

        sprintf (buf, "arg%d", ++param_count);
        $<id>$ = new IDLAstIdNode (buf);
    }
;

type:
    type_id type_pointer
    { $<tp>$ = new IDLAstTypeNode ($<id>1, $<intval>2); }
;

type_id:
    ID
    { $<id>$ = new IDLAstIdNode ($<string>1); }
;

type_pointer:
    pointer
    { $<intval>$ = $<intval>1; }
    |
    { $<intval>$ = 0; }
;

pointer:
    STAR
    { $<intval>$ = 1; }
    | STAR pointer
    { $<intval>$ = $<intval>2 + 1; }
;

array_bound:
    bound
    { $<bound>$ = $<bound>1; }
    |
    { $<bound>$ = NULL; }
;

bound:
    ARRAYBOUND
    { $<bound>$ = new IDLAstBoundNode ($<string>1); }
    | bound ARRAYBOUND
    {
        $<bound>1->addBound ($<string>1);
        $<bound>$ = $<bound>1;
    }
;

%%

void
yyerror (const char *s)
{
    printf ("Error: %s\n", s);
}

```