CITI Technical Report 93–4

An Environment for "Sniffing" DCE-RPC Traffic

James Howe
James.Howe@umich.edu

ABSTRACT

The ability to watch the network traffic generated by client and server applications can greatly assist in both understanding how a client/server application functions, as well as identifying problem areas. At the Center for Information Technology Integration (CITI), we use the Open Software Foundation's Distributed Computing Environment (OSF/DCE). To examine network traffic in this client/server environment, we developed two tools: one based on the Network General Sniffer, and one based on the IBM RS 6000 AIX operating system.

June 21, 1993

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

An Environment for "Sniffing" DCE-RPC Traffic

James Howe

June 21, 1993

Introduction

The ability to watch the network traffic generated by client and server applications can greatly assist in both understanding how a client/server application functions, as well as identifying problem areas. Over the years, many different hardware and software tools have been developed that allow users to watch network traffic. Two such tools are the focus of this paper, the Network General Sniffer and the iptrace/ipreport programs available on the IBM RS/6000 running the AIX operating system.

The Network General Sniffer is a modified IBM PC compatible computer that runs a special software package. The Sniffer allows a user to capture network traffic and then display the traffic in human readable form. The basic Sniffer has the capability to interpret and display several network protocols including IP, TCP, UDP, X Window System, and AppleTalk. In addition, the Sniffer provides the capability to add new protocol interpreters. This feature allows any organization to write an interpreter for any arbitrary network protocol.

The iptrace/ipreport programs come with the standard AIX 3.x operating system. The iptrace program collects network traffic to and from the machine and ipreport displays the information contained in the trace file. Unlike the Sniffer, the iptrace/ipreport programs do not offer any "hooks" that give a user the ability to arbitrarily add new protocol interpreters. Altering the operation of ipreport requires access to the program's source code.

The Sniffer has a broader application; it can capture all traffic on a particular sub-network and it provides a simple and easy-to-use user interface. Iptrace/ipreport, on the other hand, can only monitor traffic coming and leaving the particular machine where the program is running. Also, the output of ipreport can be quite large and sifting through the information can be difficult. However, the program can be run on any RS/6000 so more users can potentially use the program at the same time, whereas the Sniffer can only be used by one person or group at a time.

The Goal

At the Center for Information Technology Integration (CITI), we use the Open Software Foundation's Distributed Computing Environment (OSF/DCE). Because DCE is heavily based upon client/server computing, we needed a tool to monitor the information produced by DCE application and server programs. The ability to actually see the conversations occurring between a client and a server is invaluable for finding and fixing communication-related bugs. The basic goal was to decode DCE Remote Procedure Call (RPC) packets and display the information contained in the packet in a human readable form. In realizing that goal, we created two tools: one included a new Sniffer module and modifications to the DCE Interface Definition Language (IDL) compiler, and the other a new RS 6000 interpreter module for the ipreport program.

Problems

Impediments to our goal included problems identifying DCE-RPC packets and interpreting data, as described in the following sections.

Identifying DCE Packets

When a DCE packet is sent from one machine to another, the DCE-RPC mechanism bundles the information to send and uses a standard protocol such as UDP or TCP to send the packet. The receiving machine decodes the packet after it receives it. Although the Sniffer can eavesdrop on this conversation, it has no tools for identifying or decoding the contents of DCE packets.

Possible methods of identifying DCE packets include providing the Sniffer with the addresses of specific machines or communication ports on which DCE-RPC traffic will occur. However, while it may be true that each of the machines is sending or receiving DCE-RPC traffic, there is no guarantee that all, or even a high percentage, of the TCP or UDP traffic will be DCE related. The machines could also be functioning as telnet servers, ftp servers, etc. The probability of misinterpreting a packet would be very high using this approach.

Providing the Sniffer with the port assignments would guarantee DCE-RPC traffic. However, because the ports are assigned dynamically, they vary from one Sniffer session to the next. (It might be possible to have the Sniffer actually query the DCE Endpoint Mapper, but this would almost imply making the Sniffer a DCE client-a significant effort)

For our solution, we wanted to avoid the burden of requiring a user to provide specific site configuration information. It is possible to identify DCE datagram packets with a high degree of certainty. When RPC calls are made in a connectionless environment, each packet contains sufficient information for identifying itself as a DCE-RPC packet. Each RPC interface definition is identified by a unique user ID known as a UUID, a 64-bit value that is guaranteed to be unique. Every datagram packet has a header which contains the interface UUID at a known location. An RPC packet can be identified by comparing the 64-bit value at the known location with a table of interfaces known to the Sniffer. This table could be read in from a file, but additional capabilities are possible if the table is linked into the actual Sniffer executable module. In our solution, the table points to subroutines that are able to properly decode the parameter data contained in the packet.

Although it is fairly easy to identify RPC packets sent via datagrams, this is not the case for packets sent via connection-oriented mechanisms. Because the client and server establish a connection, information identifying the conversation can be sent at the start of the connection and subsequent packets can simply refer to some connection handle to properly process the packet. Because the Sniffer is eavesdropping, it may miss the initial handshake and is be unaware of the connection UUID used for the transmission. Properly identifying connection-oriented RPC packets remains problematic.

Data Interpretation

After a packet is identified as an RPC packet, there must be some way to decode the information contained in the packet. All RPC packets contain some header information followed by data. Header information is easy to interpret, it has a constant format. Data, however, varies from interface definition to interface definition. One of the desired goals for the Sniffer project was the ability to decode the data contained in an RPC packet, in addition to header information. The data portion of a packet contains the various RPC parameter values.

When an RPC is made, all of the parameters from the sending machine are “marshalled” out onto the network as a string of bytes. The receiving machine reads these bytes and “unmarshalls” the bytes back into valid data types. To decode an arbitrary packet, the Sniffer must know what the parameter definitions are for each interface it must decode. For simple variables, such as integers and strings, the decoding is fairly simple. Unfortunately, the DCE-RPC Interface Definition Language allows a user to define arbitrarily complex data types. These data types can require large amounts of code to interpret.

One particularly difficult piece of data to interpret is the “pipe.” A pipe is simply an open-ended sequence of bytes in some format that is known to both the sender and receiver. When the client and server code stubs are generated, the application developer must specify user written routines that are to be called when pipe data is processed. The RPC layer will process normal parameters and when it encounters a pipe parameter, it will repeatedly invoke the user routines to read the pipe data. Eventually, the user routine will indicate that the pipe is empty and the RPC layer will process the remaining parameters. As an outsider, the Sniffer does not have access to the user written code that is needed to properly process the data in the pipe. Because the pipe data can be in any format, it is not possible for the Sniffer to simply skip over the pipe data and continue processing. Because the Sniffer does not know the format of the data in the pipe, it cannot find the end of the pipe data. If it cannot find the end of the pipe, it cannot accurately interpret any remaining parameters.

Implementation Details

The work to decode DCE-RPC packets consists of modifications to the DCE Interface Definition Language (IDL) compiler and construction of an interpreter module for the Sniffer and for the ipreport program.

IDL Compiler Modifications

Developers use the IDL compiler to write distributed applications using DCE RPC. A developer defines each remote procedure using a language known as IDL. The definition specifies the names of remote procedures and their parameter definitions. Each interface is associated with a unique UUID. Given that each RPC definition is unique, the only way to accurately decode a packet is to have knowledge of each packet’s contents, based on the interface definition. Because the IDL compiler is working with the interface definition, it seemed reasonable that it should be the one to produce the information needed by the Sniffer.

For proper interpretation of a datagram RPC packet, the Sniffer interpreter module needs to be able to associate an interface with a set of decoding procedures that can properly interpret the contents of RPC packets belonging to a particular interface. The IDL compiler was modified to produce two files; an interface definition and a file containing interpreters for each operation defined by the interface. For example, if we had an RPC definition that looked like this:

```
[
uuid(007E7052-0735-19AD-B1E2-02608C2C832B),
version(1.1)
]
interface binop
{

[idempotent]
void binop_add
(
    [in] handle_t h,
    [in] long a,
    [in] long b,
    [out] long *c,
    [out] long *d
);

[idempotent]
void binop_mult
(
    [in] handle_t h,
    [in] long a,
    [in] long b,
    [out] long *c
);

lpq

}
```

The interface definition would look like this:

```
/* Generated by IDL compiler version DCE 1.0 */
#include "dcesniff.h"

extern OP_RECORD dump_binop_1_1[];

INTF_DEF intfArray[] =
{
    {{0x00, 0x7e, 0x70, 0x52, 0x07, 0x35, 0x19, 0xad, 0xb1, 0xe2,
                    0x02, 0x60, 0x8c, 0x2c, 0x83, 0x2b},
                    0x00010001,
     "binop",
     (OP_RECORD *) dump_binop_1_1}
};

int numInterfaces = 1;
```

and the stub file containing routines to process each operation would look like this:

```
/* Generated by IDL compiler version DCE 1.0 */
#include "dcesniff.h"

static void dump_binop_add(dir, data, len, frag, drep)
int dir;
char *data;
int len;
int frag;
ndr_format_t drep
{
```

```
    char *outBuffer;

    outBuffer = pif_line(len);
    if (outBuffer == NULL)
                    return;

    if (frag)
    {
                    genericDump(dir, data, len);
                    return;
}
    /* Handle arguments going to the server */
    if (dir == IN_DIR)
    {

rpc_op_t op;
rpc_mp_t mp;
IoVec_t(1) sp_iovec;
unsigned long space_in_buffer;

/* local variables */
idl_long_int *d_;
idl_long_int *c_;
idl_long_int b_;
idl_long_int a_;
idl_long_int IDL_ptr_1;
idl_long_int IDL_ptr_0;
if(((*drep).int_rep==ndr_g_local_drep.int_rep))
{
/******* Marshalling a_ *******/
rpc_unmarshall_long_int(mp, a_);
rpc_advance_mp(mp, 4);
/******* Marshalling b_ *******/
rpc_unmarshall_long_int(mp, b_);
}
else
{
/******* Marshalling a_ *******/
rpc_convert_long_int((*drep), ndr_g_local_drep, mp, a_);
rpc_advance_mp(mp, 4);
/******* Marshalling b_ *******/
rpc_convert_long_int((*drep), ndr_g_local_drep, mp, b_);
}
if (elt->buff_dealloc) (*elt->buff_dealloc)(elt->buff_addr);

}
    /* Handle arguments coming back from the server */
    if (dir == OUT_DIR)
    {

rpc_op_t op;
rpc_mp_t mp;
IoVec_t(1) sp_iovec;
unsigned long space_in_buffer;
if((drep.int_rep==ndr_g_local_drep.int_rep))
{
/******* Marshalling (*c_) *******/
rpc_unmarshall_long_int(mp, (*c_));
rpc_advance_mp(mp, 4);
/******* Marshalling (*d_) *******/
rpc_unmarshall_long_int(mp, (*d_));
```

```
}
else
{
/******* Marshalling (*c_) *******/
rpc_convert_long_int(drep, ndr_g_local_drep, mp, (*c_));
rpc_advance_mp(mp, 4);
/******* Marshalling (*d_) *******/
rpc_convert_long_int(drep, ndr_g_local_drep, mp, (*d_));
}
if (elt->buff_dealloc) (*elt->buff_dealloc)(elt->buff_addr);
}
}

static void dump_binop_mult(dir, data, len, frag, drep)
int dir;
char *data;
int len;
int frag;
ndr_format_t drep
{
    char *outBuffer;
    outBuffer = pif_line(len);

    if (outBuffer == NULL)
                   return;
    if (frag)
    {
                   genericDump(dir, data, len);
                   return;
    }
    /* Handle arguments going to the server */
    if (dir == IN_DIR)
    {
rpc_op_t op;
rpc_mp_t mp;
IoVec_t(1) sp_iovec;
unsigned long space_in_buffer;

/* local variables */
idl_long_int *c_;
idl_long_int b_;
idl_long_int a_;
idl_long_int IDL_ptr_0;
if(((*drep).int_rep==ndr_g_local_drep.int_rep))
{
/******* Marshalling a_ *******/
rpc_unmarshall_long_int(mp, a_);
rpc_advance_mp(mp, 4);
/******* Marshalling b_ *******/
rpc_unmarshall_long_int(mp, b_);
}
else
{
/******* Marshalling a_ *******/
rpc_convert_long_int((*drep), ndr_g_local_drep, mp, a_);
rpc_advance_mp(mp, 4);
/******* Marshalling b_ *******/
rpc_convert_long_int((*drep), ndr_g_local_drep, mp, b_);
}
if (elt->buff_dealloc) (*elt->buff_dealloc)(elt->buff_addr);
    }
```

```
    /* Handle arguments coming back from the server */

    if (dir == OUT_DIR)
    {
rpc_op_t op;
rpc_mp_t mp;
IoVec_t(1) sp_iovec;
unsigned long space_in_buffer;
if((drep.int_rep==ndr_g_local_drep.int_rep))
{
/******* Marshalling (*c_) *******/
rpc_unmarshall_long_int(mp, (*c_));
}
else
{
/******* Marshalling (*c_) *******/
rpc_convert_long_int(drep, ndr_g_local_drep, mp, (*c_));
}
if (elt->buff_dealloc) (*elt->buff_dealloc)(elt->buff_addr);

    }
}

/*
** Array of operation names and associated dump routines
*/

OP_RECORD

    dump_binop_1_1 [2] =
    {
                {"binop_add", dump_binop_add},
                {"binop_mult", dump_binop_mult}
    };1
```

The changes made to the IDL compiler consisted of two areas: additional option support and additional stub generation. To generate appropriate stubs, several new options were added to the compiler. They were `-sniffer`, `-intfstub`, `-snstub`, `-generic`. To have the compiler accept the new options, the following files were modified:

`backend.c`
: This file contains routines that handle the high-level, back-end processing for the IDL compiler. The changes made to the routines in this file included adding parameters to some of the functions and adding calls to the Sniffer stub generation routine.

`backend.h`
: This file contains the constants and definitions relating to functions found in `backend.c`. The changes to this file consisted mainly of altering the function definitions to allow for the additional parameters needed by some of the functions.

`command.c`
: The `command.c` file contains various declarations that define the options available. Modifications were made to the various tables to include new options.

`command.h`
: The `command.h` file contains various constants used in the procession of compiler options. Additional constants were added to support the new compiler options.

`driver.c`
: The main driver had to be modified to handle the creation of stub files generated when the Sniffer option is specified. These files are then passed to the back end of the compiler as parameters.

`hdgen.c`
: This file contains functions that are used when generating the header (function and parameter definitions) for a stub routine. Additional routines were added to generate the Sniffer stub routine.

`hdgen.h`
: This file contains the function definitions used by `hdgen.c`. It was modified to include the new functions needed to support Sniffer stub generation.

`sniffer.c`
: This file contains the primary functions that are used to generate the Sniffer stub routine. It is modeled after the `client.c` and `server.c` files.

`sniffer.h`
: This file contains the function, type, and constant definitions used by `sniffer.c`.

`snstubgen.c`
: This file contains functions that are used to generate the stub files that are bound into the Sniffer module. The routines in this file generate the basic skeleton for the stub routines, with functions to generate specific unmarshalling code being contained in other files. This is a new file, derived from code found in `sstubgen.c`.

`snstubgen.h`
: This file contains the function definitions used by `snstubgen.c`. This is a new file.

`snunmarsh.c`
: This file contains routines that will unmarshall parameters in a form compatible with the Sniffer. This is a new file but was derived from code found in `munmarsh.c`.

`snunmarsh.h`
: This file contains the function definitions used by `snunmarsh.c`. This is a new file.

`sysdep.h`
: This file contains various system dependent declarations. New declarations were added to support the additional stub routines that are generated by the Sniffer option.

Sniffer Modifications

The other component of the interpreter is a new interpretation module used by the Sniffer when interpreting RPC packets. Because we are looking for packets transmitted via UDP or TCP, a module replacing the standard Sniffer UDP/TCP interpreter was needed. The Sniffer design makes it easy to add custom interpreters. A master definition file gives the developer the ability to “plug in” his or her own interpreter in place of standard intepreters. In our case, we needed to replace the IP interpreter that handles the interpretation of both UDP and TCP. Because our module would see all TCP and UDP traffic, including traffic that was not DCE-RPC related, we needed to write the module in such a way that non-DCE packets would be properly interpreted.

Our Sniffer module is designed to handle RPC packet headers. When further interpretation of the packet is desired, the module calls the routines defined in the stub code. When the Sniffer module is built, the generic code is combined with the code produced by the IDL compiler to create a module that has the ability to interpret interfaces defined by the interface and stub code. The basic algorithm used by the generic DCE-RPC interpreter is as follows:

> Look for UDP or TCP packets. If the packet is not UDP or TCP, call the standard IP interpreter.
>
> If the packet is a TCP packet, check the source and destination address for a “known” DCE server address (contained in a config file). If a match is found, interpret the packet by displaying the contents of the header. If a match is not found, drop out and let normal TCP processing take over.
>
> If the packet is a UDP packet, check to see if it is an interface found in the interface table. If it is, set up a pointer to the interface definition. If it is not, check to see if the packet has come from a “known” machine. If it has, and the packet is big enough to be an RPC packet, and it has a version number that matches a known value, process the packet as an “unknown” interface. Otherwise drop out, and let normal UDP processing take over.
>
> If a valid RPC packet is found, display the contents of the header. If the packet is from a “known” interface, dump the contents of the packet using the routines contained in the stub file. The proper routine is found by referencing a subroutine contained in a table indexed by the operation number.

The changes made to the Sniffer consisted of modifying a routine supplied with the Sniffer, as well as writing a new packet interpretation routine. Some auxiliary Sniffer files were also modified to add new menu options to the Sniffer menu.

`initpi.c`
: This file is supplied with the Sniffer. It allows a developer to add new protocol interpreter definitions, as well as select which interpreter definitions are active. We modified this file to reference our IP packet interpreter code instead of the code supplied with the Sniffer. See the Network General Sniffer programming documentation for more information about this file.

`dceip.c`
: This file contains our IP protocol interpreter. One of the first things this code does is check to see if the packet received should be processed by our interpreter. If it does not meet certain qualifications (discussed above), the packet is simply sent to the standard interpreter for processing.

`dcesniff.h`
: This header file contains definitions used by `dceip.c`. This is a new file.

`xxstub.c`
: This file is generated by the IDL compiler and will vary from IDL definition to IDL definition. It is compiled and linked with other Sniffer-related object modules and libraries to create a Sniffer module that can process RPC packets generated by programs using a particular IDL specification.

Ipreport Modifications

An alternative method of capturing and viewing DCE-RPC packets is available through the ipreport program provided with AIX. This program processes a log file created by the iptrace program and decodes various protocols. Making simple changes to the basic ipreport program, we were able to have it identify and decode DCE-RPC packets. We changed the main ipreport program to call our decoding function when an IP packet was being processed, much the same way the Sniffer works. In fact, our decoding routine is the same routine used in our Sniffer implementation. We wrote a separate library that emulated some of the routines provided by the Sniffer package. The specific RPC decoding routines are generated by the IDL compiler, and once again, are the same ones used when building the Sniffer module.

The following changes were made to support DCE-RPC packet decoding with the AIX ipreport program:

`ipreport.c`
: This file was modified to invoke our IP interpretation routine. If our routine does not process the packet, the standard decoding routine provided by ipreport is used.

`piflib.c`
: This file contains functions that emulate functions provided by the standard Sniffer libraries. These functions allow us to use a common decoding layer for both the Sniffer module and the ipreport program.

`pi.h`
: This file contains definitions used by the protocol decoding routine. The Sniffer provides a version of this file for building Sniffer modules. We needed to create our own version for use with ipreport, so that we could use a common protocol decoding routine.

Future Work

The changes that were made to the Sniffer and the ipreport programs have proven useful in capturing and examining DCE-RPC traffic. However, improvements are always needed. The following sections describe some additional capabilities that would improve our DCE-RPC packet "sniffer."

Improved Connection-Oriented Interpretation

The current implementation of the DCE-RPC Sniffer module recognizes connection-oriented RPC traffic by examining the source or destination of the packet and comparing the address with a list of addresses that are likely to generate DCE traffic. If a packet is associated with a known DCE machine, and the packet is large enough to be a valid RPC packet, the information in the packet is decoded. Unfortunately, only the header information can be decoded, and the connection-oriented packet does not contain much useful information.

We would like the Sniffer to obtain RPC interface information based on the connection ID found in the packet header. Because the connection ID is dynamically assigned, we need a mechanism in which the Sniffer is able to locate the original connection request and see which interface was associated with a particular connection ID. It could save this information for later use. Of course, the Sniffer would also have to know when the connection was closed, so it could discard this information. Because a user can capture packets after a connection-oriented conversation has started, not all such conversations could be completely decoded. It should be possible, however, to decode conversations that are contained within captured data.

Add Support for Interface Definitions from a File

Currently, the Sniffer module understands and recognizes packets that have UUID's which are bound into the program module. The Sniffer can also decode probable RPC packets that go to or from particular machines identified in the configuration file. Unfortunately, the interface name and the name of individual operations are not given, and simply appear as "<unknown>". It would be possible to expand the configuration file definition to include the capability to define a list of known interfaces and their operations. This would require a file format that could associate an interface name with a UUID, and that could associate one or more operations with an interface name.

For this scheme to be useful, the file would have to be able to contain more than one interface definition. This would simplify the incorporation of new interfaces (either because new ones are identified, or changes have been made to existing ones) but would increase the complexity of configuration file processing. Using the bound interface definition has the advantage of being able to make use of subroutines to decode packets belonging to operations, but for cases where only the header information is needed, specifying information in the config file would be useful.

Improve Host Machine Support

Currently, RPC traffic can be captured and analyzed using either the Sniffer, or the iptrace/ipreport programs that reside on an RS/6000. The Sniffer provides an easy-to-use interface that lets a user control what information is captured and what information is displayed. The ipreport program, on the other hand, simply dumps all packet information to a file. There are no filters than can be specified that limit the output. A simple enhancement could be made to the ipreport program that would let a user specify some parameters to control the amount and type of output generated. A more complicated, but more user friendly, change would be to create an X Window System based program from which the user could control and view the output of the program.

Implement a "Mini-Sniffer"

It should be possible to implement a subset of the features found on the Sniffer for a PC-based operating system such as OS/2. This would have the advantage of not requiring access to the Sniffer machine. In particular, we would only be concerned with capturing traffic and decoding TCP/IP packets.

Conclusion

The modifications made to enable the Sniffer and the ipreport programs to capture and decode DCE-RPC packets have been useful to the DCE work done at CITI. The ability to capture and analyze DCE traffic is an invaluable aid to understanding how DCE functions and in identifying bugs in client/server applications, including the core DCE services. In the future, we hope to be able to use captured information for performance analysis and tuning.

Acknowledgements

Thanks to Mary Jane Northrop and Mike Stolarchuk for their reviews of this paper. This work is partially supported by IBM.