CHAPTER VIII
OPERATIONS &
CONCEPTS USED IN THE
NETWORK PROGRAMMING
The Operations and Concepts used in the Network Programming

This chapter discusses the operations and concepts used in the Programming of Network management functions. The objective of this discussion is to show that management functions can be developed together with primary functions as part of the same process.

The structure of this chapter is as follows. Section 8.1 gives a short introduction about Network Programming. It also discusses some basic functions like t_alloc, unbind, t_close, t_open etc.

Section 8.2 describes the Management of Distributed databases. In DD the information is stored on several computers. The computers in a distributed system communicate with each other through various communication medias based on some protocols. Section 8.3 gives you the guidelines to compile your RPC and using its various layers for applications.

Section 8.4 discusses Data Transmission by using the XDR protocol. This particular section will explain some routines that allow a C program to describe arbitrary data structures in a machine independent fashion. Section 8.5 gives you the Prementation details for a family of XDR streams-creation routines exists in which each member treats the stream of bits differently. Section 8.6 describes Non-filter primitives and section 8.7 gives you the idea about XDR operation directions and section 8.8 gives XDR streams implementation. This section provides the abstract data types needed to implement new instances of XDR streams. In Section 8.9 we will discuss about the two application program interfaces (APIs) XTI and TLI. Section 8.10 based on addresses of Transport for data communication and next section 8.11 gives you a idea about types of services followed by general discussion about Synchronous and Asynchronous Operation.

8.1 What is Network Programming

The basic function or responsibility of network programming is to manage the communication between processes that is executing on different computers. Here we will use the "client- server" model. In this model, a process on the computer (the server) waits for other processes on other computers (the clients) to contact it [server]. If the server accepts the connection from a given client, it can perform some service that may be some application or computation from a given client.

The above usage is more precise than the common practice of confusing a server or client process with the computer on which it is running. For example, the term
"file server" is often used to refer to the computer on which the file server process is running. However, the same computer can run not only the file server process, but also several other kinds of servers.

Similarly, a computer on which one kind of client is running can also run many other clients. In fact, a server process on one computer can, in order to perform its task on behalf of the client that contacted it, enlist the assistance of yet another server on a different computer. It does the using the same mechanisms that the client on the first computer used. The result is that, while the first server is communicating with the second server, it is actually functioning as a client.

In this model, server processes can be running on many different computers that are attached to many different networks. Whenever a process need some service or computation performed that it either cannot, should not, or need not do, it simply contacts the appropriate server (wherever it is on the network). Moreover, servers can connect to each other to request services or computations. The entire network then becomes an interconnected set of building blocks that can be called upon in an organized fashion to accomplish complex tasks.

To contact a particular server, a client first creates a transport endpoint by calling system function \texttt{t_open}. In this function call, the client specifies the transport provider that will be used by the client. The client then calls \texttt{t_alloc} to allocate a data structure to contain addresses information. The client then calls \texttt{t_connect} to contact the server, passing the data structure to the information as an argument.

Before clients can successfully connect to a server, the server must be waiting for an incoming connection request. The server first calls \texttt{t_alloc} to allocate a data structure to contain address information. It then calls \texttt{t_listen} and waits for a client to contact it. When a connection request arrives from client, \texttt{t_listen} returns with the client address information in the allocated address information in the allocated data structure. The server can either reject the request by calling \texttt{t_snddis} (send disconnect) or accept the request by calling \texttt{t_accept}. The acceptance and rejection of the request is depending on the server.

When a client is finished communicating over the network, it calls function \texttt{t_snddis} to abort the connection, function \texttt{t_unbind} to disassociate address information from the transport endpoint, then function \texttt{t_close} to delete the transport endpoint. If the client is simply finished with the one server and wants to connect to another, it calls \texttt{t_snddis} to abort the connection, followed by \texttt{t_connect} to send a connection request to the new server.

8.2 Management of Distributed Databases
In a Distributed databases, the information is stored on several computers. The computers in a distributed system communicate with each other through various communication medias based on some protocols.

ONC-RPC (Remote Call) was developed to facilitate the development of distributed client/server applications. Using RPC, We can write distributed applications in nearly the same way they write non-distributed application today.

In RPC, the caller process causes the server process to execute a procedure call, much as if the calling code and the called procedure run as two separate processes, so they do not have to execute on the same physical machine. The RPC mechanism is implemented as a library of procedures, plus a specification for portable data transmission known as XDR (External Data Representation). Both RPC and XDR are portable, providing a standard I/O library for interprocesses communication, either on one machine or across a network. The rpcgen (NC) utility automatically generates header files and stubs linked into the client and server code to transparently perform the operations required implementing RPC and XDR.

8.3 How to Compile RPC

When you write an application that uses the RPC protocol to call a program on a remote machine we can use any language [here we are using C language] called Remote Procedure Call Language (RPCL). Once the code is written in RPCL, use the rpcgen (NC) command to generate actual language code that implements the remote procedure call in accordance with the RPC protocol. The RPCL input may contain C-style comments and normal C language preprocessor directives. Comments are simply ignored, and the directives are copied uninterpreted into the output header file.

When creating XDR routines, you can customize them by leaving some of the data types undefined. When rpcgen encounters these undefined data types, it assumes the existence of a corresponding routine named xdr_type_name, where type_name is the name of the undefined data type. Using the various options, you can compile XDR routine, compile C data definitions (a header file)), specify the name of the output file, or compile a server using the given transport.

8.3.1 How to Use RPC

Programs that communicate over a network need a paradigm for communication. The method used by the NFS is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a request to the server. When the packet containing the request
arrives, the server calls a dispatch routine, performs the service requested, sends back the reply, and the procedure call returns to the client.

The RPC interface is divided into three layers.

The highest layer is totally transparent. At this level, a program can contain a call to `rnusers`, which returns the number of users on a remote machine.

The middle-layer routines are designed for most common applications and shield you from needing to know about sockets.

The lowest layer routines are designed and developed when we are sending more than 8K of data or for allocation and deallocation of memory. You can also use this layer for user authentication.

At this layer, the routine `registerrpc` and `callrpc` are used to make RPC calls: `registerrpc` obtains a unique system-wide number, while `callrpc` are used to make RPC call: `registerrpc` obtains a unique system-wide number, while `callrpc` executes a remote procedure call.

The `rnusers` call can be implemented by using the following two routines.

### 8.3.2 Using the Highest Layer

Suppose we are writing a program that needs to know how many users are logged into a remote machine. We do this by using the routine named `rnusers` given here:

```c
#include <stdio.h>
main (argc,argv)
    int argc ;
    char **argv;
{
    unsigned num;
    if ( argc < 2 ) {
        fprintf (stderr, "usage: rnusers hostname\n")
        exit (1);
    }
    if ( ( num = musres (argv[1]) ) < 0 ) {
        fprintf (stderr, "error : rnusers\n");
        exit (-1);
    }

```
printf ("% users on %s\n", argv[1]);
}
printf ("%d users on %s
", num, argv[1]);

The program above could be compiled with:

cc program.c -lrpcsvc -isocket

8.3.3 Using the Intermediate layer

The simplest interface, which explicitly makes RPC calls, uses the functions callrpc and registerrpc. Here is the alternative way to get the number of remote users:

```c
#include <stdio.h>
#include <utmp.h>
#include <rpc/types.h>
#include <rpc/xdr.h>
#include <rpcsvc/rusers.h>

main (argc, argv)
    int argc;
    char ** argv;
{
    unsigned long nusers;
    if (argc < 2) {
        fprintf (stderr, "usage: nusers hostname\n");
    }
    if (callrpc (argv[1], RUSERSPROGL, RUSRSPVERW,
                 RUSERSPROC_NUM, xdr_void, 0, xdr_u_long, &nusers) != 0)
    {
        fprintf (stderr, "error: callrpc\n");
        exit (1);
    }
    printf ("number of users on %s is %ld\n", argv[1], nusers);
    exit (0)
}
```

A program number, version number, and procedure number define each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (for example, adding a new procedure) a new program number does not have to be assigned.
When you call a procedure to find the number of remote users, the appropriate program, version and procedure number are looked up in manual, in a similar manner to looking up the name of the memory allocator when memory is to be allocated.

The simplest routine in the RPC library used to make remote procedure calls is callrpc. It has eight parameters:

- The first parameter is the name of the remote machine.
- The next three parameter are the program, version, and procedure numbers.
- The following two parameters define the argument of the RPC call.
- The final two parameters are for the returns value of the call.

if callrpc completes successfully, it returns zero; nonzero otherwise. The exact meaning of each return code is found in <rpc/clnt.h>, and each return code is in fact an enum clnt_stat cast into an integer.

Because data type may be represented differently on different machines, callrpc needs both the type of the RPC argument and a pointer to the argument itself, and needs similar information for the result. For RESERSPROC_NUM, the return value is an unsigned long. This means that callrpc has xdr_u_long as its first return parameter, which says that the result is of type unsigned long, and has & nusers as its second return parameter, which is a pointer to where the long result will be placed. Because RSERPRC_NUM takes no argument, the argument parameter variable is NULL.

The callrpc procedure uses the user Datagram Protocol (UDP) to send a message over the network and wait for a response. If UDP receives no responses, it again sends the message and waits for a response. After trying several times to deliver a message and receiving no response, callrpc returns with an error code. Methods for adjusting the number of retries or for using a different protocol require the use of the lower layer of the RPC library.

The procedure takes one argument, which is a pointer to the input of the remote procedure call and it returns a pointer to the result. In the following program, character pointer are the generic pointers, so both the input argument and the return value are cast to char *. Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests.

In this program, there is only a single procedure to register, so the main body of the server will be:

```c
#include <stdio.h>
#include <rpcsvc/ruser.h>

char *nuser();
```
main()
{
    registerrpc (USERPROG,USERVERS,USERSPRC_NUM, nuser,
              xdr_void, xdr_u_long);
    svc_run(); /*never returns */
    fprintf(stderr, "Error:svc_run returned!
");
    exit(1);
}

The registerrpc routine establishes which C procedure corresponds to each RPC procedure number. The parameters are:

- The first three parameters, USERPROG, USERVERS, and USERSPRC_NUM, are the program, version, and procedure numbers of the remote procedure to be registered.
- The name of the Procedure that implements it is nuser.
- The types of input and output from the procedure.

8.3.4 How to assign program numbers

Program numbers are assigned in groups of 0x20000000(536870912) according to the following chart:

0-1fffffff defined by System Software[Supplier of the software i.e. Sun, SCO etc.]
20000000 - 3fffffff defined by user
40000000 - 5fffffff transient
60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved

- The first group of numbers is administered by system. The intent is that they will be identical across all systems and applications. If a programmer develops an application that might be of general interest, that application should be given a number assigned by Sun from the first range.
- The second group of numbers is reserved for specific applications. This range is intended primarily for debugging new programs.
• The third group is reserved for applications that generate program numbers dynamically.

• The final groups are reserved for future use, and should not be used.

8.3.5 Using XDR to pass arbitrary data types

In the previous section, the RPC call passes a single unsigned long. RPC can handle arbitrary data structures, regardless of the byte order or structure layout conventions of different machine architectures. It does this by always converting, the data to a network standard, external Data Representation (XDR), before sending the data over the wire. The process of converting from a particular machine representation to XDR format is called serializing, and the reverse process is called deserializing. The type field parameters of callrpc and registerrpc can be a built-in procedure like xdr_u_long in the previous program or a user-supplied one. XDR has these built-in type routines:

```
xdr_int()  xdr_u_int()  xdr_enum()
xdr_long() xdr_u_long() xdr_bool()
xdr_short() xdr_u_short() xdr_string()
```

As an example of a user-defined type routine, assume that you want to send the following structure:

```
struct simple {  
    int a;  
    short b;  
} simple;
```

Then, callrpc should be called as

callrpc (hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);

where xdr_simple is written as:

```c
#include <rpc/rpc.h>
xdr_simple (xdrsp, simple)
xdr * xdrsp;
struct simple *simple;
{
    if (! xdr_int(xdrsp, &simple->))
        return (0);
    if (! xdr_short(xdrsp, &simple->b))
        return (0);
```

182
An XDR routine returns nonzero (TRUE in the sense of C) if it complete successfully, and zero otherwise. In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()  xdr_bytes ()
xdr_reference()  xdr_union()
```

To send a variable array of integers we can include them in a structure like this:

```
struct varintarr {
    int *data;
    int arrlnth;
} arr;
```

and make an RPC call such as:
```
callrpc (hostname, PROGNUM,VERSNUM,PROCNUM,xdr_varintarr, &arr...);
```

with `xdr_varintarr` defined as:
```
xdr_varintarr(xdrsp,varintarr)
    XDR *xdrsp;
    struct varintarr * arrp;
{
    return(xdr_array(xdrsp,&arrp->data, &arrp->arrlnth, MAXLEN,
                     sizeof (int), xdrint));
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element. If the size of the array is known in advance, then the following could also be used to send out an array of length SIZE.

```
int intarr [SIZE ] ;

xdr_intarr (xdrsp, intarr)
    XDR *xdrsp ;
    int intarr [ ];
{
    int i ;
    for (i = 0; i < SIZE; i++)
    {
if (!xdr_int(xdrsp, &intarr [i]))
    return (0);`
XDR always convert quantities to 4-byte multiples when deserializing. Thus, if either of the program above involved character instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine xdr_bytes without the length parameter. On serializing, it gets the string length from strlen and on deserializing, it creates a null-terminated string.

Here is a final program that calls the previously written xdr_simple as well as the built-in functions xdr_string and xdr_reference, which chases pointers:

```c
struct final {
    char *string;
    struct simple *simplep;
} final;
xdr_final (xdrsp,final)
XDR *xdrsp;
struct final *finalp;
{
    int i;
    if (!xdr_string (xdrsp,&finalp->string,MAXTRLEN ) )
        return ( 0 ) ;
    if (!xdr_reference (xdrsp, &final ->simplep,
        size of (struct simple ), xdr_simple) )
        return (0) ;
    return (1) ;
}
```

8.3.5 Using the Lower layer

It is advised to avoid using the lower layers of RPC. If in any case the use of lower layer is must then in order to perform the following task you may use the lower layers

- Want to send more than 8k bytes of data.
- Allocation and deallocation of memory
- In order to perform authentication.

We conclude that each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from callrpc, the serializing part is used. When called from svc_getargs, the deserializer is used. When called from svc_freeargs, the memory deallocator is used.
8.4 Data Transmission by using the XDR protocol

This particular section will explain some routines that allow a C program to describe arbitrary data structures in a machine independent fashion. The eXternal data representation (XDR) standard is the backbone of the rpc’s is transmitted using the standard XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This section focuses on accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures.

8.4.1 Compiling XDR routines

We have to include the header file <rpc/rpc.h> contains all the interfaces to XDR.

Program to Create portable data with XDR <given in annex>

The two programmes appear to be portable for the following reasons:

- They pass lint checking.
- They exhibit the same behavior when executed on different hardware architectures.

The behavior of these two programmes can be verified by piping the output of the writer program to the reader program. For example, these programmes should produce identical results when run on both an SCO system and a sun workstation, as shown in the examples below.

Nix% writer/reader
01234567
nix %

sun % writer/reader
01234567
sun %
With the advent of local area networks came the concept of network pipes, in which a process on one machine produces the data, and a second process on a different machine consumes the data. A network pipe can be constructed with writer and reader. Below is an example of a network pipe in which an SCO system produces data and a sun workstation consumes the data.

```
Nix % writer / rsh sun reader
016777216 33554432 50331648 6710886080 83886080 100663296
117440512
nix%
```

If the machine on which each program is run is changed, the result will be the same. These results occur because the byte ordering of long integers differs between these machines. Other data type can have varying sizes, byte orderings, representatives, and alignments, depending on the underlying hardware of the machine. For example, the number 01234567 is stored on an SCO system as follows:

<table>
<thead>
<tr>
<th>BYTE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>67</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
</tr>
</tbody>
</table>

A sun stores the same number in the following way:

<table>
<thead>
<tr>
<th>BYTE</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>67</td>
</tr>
</tbody>
</table>

Note that 16777216 is $2^{24}$; when four bytes are reversed, the 1 winds up in the 24th bit.

This example shows the need for portable data, a need exists whenever data is shared by two or more machine types. Programs can be made data portable by replacing the `read()` and `write()` call with calls to an XDR library routine `xdr_long()`. This routine is a filter that knows the standard representation of a long integer in its external form.

Here the results from executing the new programs in three different ways: both programs on an SCO system, both programs on a sun workstation, and one program on each machine:

```
Nix% writer / reader
01234567
```
Dealing with integers is only a small part of portable data. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but have no outside where they are defined.

The XDR library package solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter for many subjects, including strings (null terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write you're own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves mat contain arrays of arbitrary elements or pointers to other structures.

8.5 Prementation details

A family of XDR streams-creation routines exists in which each member treats the stream of bits differently. In the example given, data is manipulated using standard I/O routines, so xdrstdio_create() is used. The parameters to XDR stream-creation vary according to their function.

In the example, xdrstdio_create() takes a pointer to an XDR structure that it initializes, a pointer to a file that the input or output is performed on, and the operation. The operation may be XDR_ENCODE for serializing in the writer program or XDR_DECODE for deserializing in the reader program. RPC clients never need to create XDR streams. The RPC system creates these streams, which are then passed to the clients.

The XDR_long() primitive is characteristic of most XDR library primitives and all client routines.

- The routine returns TRUE (1) if it succeeds, and FALSE (0) if it fails.
• For each data type, xxx, there is an associated XDR routine of the form:

\[
\text{Xdr}\_\text{xxx}\ (\text{xdrs, fp})
\]
\[
\text{Xdr} ^*\text{xdrs};
\]
\[
\text{Xxx} ^*\text{fp};
\]

In this case, xxx is long, and the corresponding XDR routine is the primitive XDR_long. The client could also define an arbitrary structure xxx, in which case the client would also supply the routine Xdr_xxx, describing each field by calling Xdr routines of the appropriate type. In all cases, the first parameter, xdrs can be treated as an opaque handle and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The intention is to call the same routine for either operation; this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself; only in the case of deserialization is the object modified. This feature is not shown in the program, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the direction can be obtained.

Consider a slightly more complicated program. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are implemented enough to warrant their own data type:

```c
Struct gnumbers {
    Long g_assets;
    Long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```c
bool_t /* TRUE is success, FALSE is failure */
Xdr_numbers (xdrs, gp);
{
    If (XDR_long (xdrs, &gp->g_assets) &&
        XDR_long (xdrs, &gp->g_liabilities))
        return (TRUE);
    return (FALSE);
}
```
The parameter $Xdr$'s is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This program also shows that the type $bool_t$ is declared as an integer whose only values are TRUE and FALSE (0). This section uses the following definitions:

```c
#define bool_t int
#define TRUE 1
#define FALSE 0
#define enum_t int /* enum_t's are used for generic enum's */
```

Using these conventions, $XDR_gnumbers()$ can be written as follows:

```c
Xdr_gnumbers (xdrs, gp)
{ 
    return (xdr_long (xdrs, &gp->g_assets) &&
            xdr_long (xdrs, &gp->g_liabilities));
}
```

8.5.1 XDR library primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, which is automatically included by `<rpc/rpc.h>`.

8.5.2 Number filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

$$[\text{Signed, unsigned}] \times [\text{short, int, long}]$$

Specifically, the six primitives are:

```c
bool_t xdr_int (xdrs, ip)
```
The first parameter, Xdrs, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it.

All routines return TRUE if they complete successfully and FALSE otherwise.

8.5.3 Floating-point filters

The XDR library also provides primitive routines for C's floating-point types:

```c
bool_t xdr_float (xdrs, fp)
    Xdr *xdrs;
    float * fp;

bool_t xdr_double (xdrs, dp)
    Xdr *xdrs;
    Double * dp;
```

The first parameter, xdrs, is an XDR stream handle. The second parameter is the address of the floating-point number that provides data to the stream or receives data from it.
All routines return TRUE if they complete successfully and FALSE otherwise.

8.5.4 Enumeration filters

The XDR library provides a primitive for generic enumeration. The primitive assumes that a C enum has the same representation inside the machine as a C integer. The Boolean type is an important instance of the enum. The external representation of a Boolean is always one (TRUE) or zero (FALSE).

```
#define bool_t int
#define FALSE 0
#define TRUE 1
#define enum_t int

bool_t xdr_enum (xdrs, ep)
    Xdr *xdrs
    enum_t *ep;

bool_t xdr_bool (xdrs, bp)
    XDR *xdrs;
    Bool_t *bp;
```

The second parameters, ep and bp, are addresses of the associated type that provides data to, or receives data from, the stream xdrs.

The routines return TRUE if they complete successfully, and FALSE otherwise.

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
Bool_t xdr_void () ; /* always return TRUE */
```

8.5.5 Constructed Type Filters

Constructed or compound data type primitives require more parameters and perform complicated functions than the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE.

191
The three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is passed or manipulated, a pointer to is used. Therefore, the XDR library defines a string to be a CHAR*m, and not a sequence of characters. The external representation of a string is different from its internal representation. Externally, strings are represented as sequence of ASCII characters, while internally they are represented with character pointers. Conversion between the two representations is accomplished with the routine xdr_string():

```
Bool_t XDR_string (xdrs, sp, maxlength)
XDR *xdrs;
Char** sp;
Uint maxlength;
```

The parameters operate as follows:

- The first parameter xdrs is the stream handle.
- The second parameter is a pointer to a string (type char**).
- The third parameter maxlength specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may set a file name may be longer than 255 characters.

The routine returns FALSE if the number of characters exceeds `maxlength`, and TRUE if it does not.

The behavior of xdr_string() is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter SP points to a string of a certain length; if it does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle.

- First, the length of the string is determinated; it must not exceed `maxlength`.
- Next, SP is dereferenced; if the value is null, then a string of the appropriate length is allocated and *SP is set to this string.
- If the original value of *SP is non-NULL, then the xdr package assumes that a target area, which can hold strings no longer than `maxlength`, has been allocated. In either case, the string is decoded into the target area.
The routine than appends a null character to the string.

In the XDR free operation, the string is obtained by dereferencing SP. It the string is not NULL, it is freed and *SP is set to NULL. In this operation, xdr_string ignores the maxlength parameter.

**Byte arrays**

Using variable-length arrays of bytes are often preferable to strings. Byte arrays differ from strings in the following ways:

- The length of the array (the byte count) is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes is the same as their internal representation.

The primitive xdr_bytes() converts between the internal and external representations of byte arrays:

```c
Bool_ xdr_bytes(xdrs, bpp, ip, maxlength)
Xdr *xdrs;
Char **bpp;
Uint *lp;
Uint maxlength;
```

The usage of the first, second and fourth parameters is identical to the first, second, and third parameters of xdr_string(), respectively. The length of the byte area is obtained by dereferencing IP when serializing; *IP is set to the byte length when deserializing.

**Arrays**

The XDR library package provides a primitive for handling arrays of arbitrary elements. The xdr_bytes() routine treats a subset of generic arrays in which the size of array elements is known to be 1 and the external description of each element is built in. The generic array primitive xdr_array() requires parameters identical to those of xdr_bytes() plus two more: the size of array elements and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```c
Bool_t xdr_array(xdrs, ap, ip, maxlength, elementsixe, xdr_element)
    XDR *xdrs;
```
Uint *ip;
Char *ap;
Uint maxlen;
Uint elementsize;
Boolt (*xdr_element) ();

The parameter AP is the address of the pointer to the array. If *AP is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets *ap to that array. The element count of the array is obtained from *IP when the array is serialized; *IP is set to the array length when the array is deserialized. The parameter **maxlength** is the maximum number of elements that the array is allowed to have; elementsize is the byte size of each element of the array. (The C function sizeof() can be used to obtain this value). The routine xdr_element is called to serialize, deserialize, or free each element of the array.

8.6 Non-filter primitives

XDR streams can be manipulated with the primitives discussed in this section:

```c
Uint xdr_getpos (xdrs)
    Xdr *xdrs;

Bool_t xdr_setpos (xdrs, pos)
    Xdr *xdrs;
    Uint pos;

Xdr_destroy (xdrs);
    XDR *xdrs;
```

The routine xdr_getpos() returns an unsigned integer that describes the current position in the data stream. The routine xdr_setpos() sets a position to pos. The xdr_destroy() primitive destroys the xdr stream. Use of the stream after calling this routine is undefined.

8.7 XDR operation directions

Sometimes, you may want to optimize XDR routines by taking advantage of the operation (XDR_ENCODE, XDR_DECODE, or XDR_FREE). The value xdr->x_op always contains the direction of the xdr operation. Programmers are not encouraged to take advantage of this information, so no example is presented here.

8.7.1 XDR stream access
An XDR stream is obtained by calling the appropriate creation routine. Such creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for the serialization and deserialization of data to or from standard i/o FILE streams, TCP/IP connections, files and memory. The section "XDR streams implementation " (page 199) describes the XDR object and how to make new XDR streams when they are required.

**Standard I/O streams**

You can interface XDR streams to standard I/O by using the xdrstdio_create();

```c
#include <stdio.h>
#include <rpc/rpc.h> /* xdr streams are part of the rpc library */
void
xdrstdio_create (xdrs, fp, x_op);
XDR *xdrs;
FILE *fp;
Enum xdr_op x_op;
```

The routine xdrstdio-create() initializes an XDR stream pointed to by xdrs. The XDR stream interfaces to the standard I/O library. Parameter fp is an open file, and x_op is an XDR direction.

**Memory streams**

Memory streams allow the streaming of data into or out of a specified area of memory;

```c
#include <rpc/rpc.h>
void
xdrmem_create (xdrs, addr, lin, x_op)
XDR *xdrs;
Char *addr;
Uint len;
Enum xdr_op x_op;
```

The routine xdrmem_create() initializes an XDR stream in local memory. The memory is pointed to by the parameter addr, parameter is the length in bytes of the memory. The parameter xdrs and x_op are identical to the corresponding parameters of xdrstdio-create(). Currently, the UDP/IP implementation of rpc uses xdrmem_create(). Complete call or result messages are built into memory before calling the sendto() system routine.

**Record (TCP/IP) streams**

A record stream is an XDR stream built on top of a record-marking standard that is built on top of an ordinary file or BSD connection interface.
# Include <rpc/rpc.h> /* xdr streams are a part of the rpc library */

Xdrrec_create (xdrs, sendsize, recvsize, iohandle, readproc, writeproc)
XDR *xdrs;
Uint sendsize, recvsize;
Char *iohandle;
Int (*readproc) (), (*writeproc) ();

The routine xdrrec_create() provides an XDR stream interface that allows for a bi-directional, arbitrary long sequence of records. The contents of the records are meant to be in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of ordinary files.

The parameter xdrs is similar to the corresponding parameter described above. The stream does its own data buffering, similar to that of standard I/O. The parameters sendsize and recvsize determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine readproc or writeproc, respectively, is called. The usage and behavior of these routines are similar to the system calls read() and write().

However the first parameter to each of these routines is the opaque parameter iohandle. The other two parameters (buf and nbytes) and the results (byte count) are identical to the system routine. If xxx is readproc or writeproc, then it has the following form:

/* Returns the actual number of bytes transferred. -1 is an error. */

Int
Xxx (iohandle, buf, len)
Char *iohandle
Char buf;
Int nbytes;

The XDR stream provides means for delimiting records in the byte stream. The primitives that are specific to record streams are as follows:

Bool_t
Xdrrec_endofrecord (xdrs, flushnow)
XDR *xdrs;
Bool_t flushnow;

Bool_t
Xdrrec_skiprecord (xdrs)
XDR *xdrs;

Bool_t
Xdrrec_eof (xdrs)
XDR *xdrs;
The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is `TRUE`, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns `TRUE`. This does not imply that there is no more data in the underlying file descriptor.

8.8 XDR streams implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR object

The following structure defines the interface to an XDR stream:

```c
struct xop {
    unsigned x_op; // Current operation
    struct {
        char *x_private; // Private to stream implementation
        char *x_base;    // Base of stream
        char *x_handy;   // Handy
    } x_private; // Private
    struct {
        char *x_public; // Public for XDR client
    } x_public; // Public
    struct {
        char *x_getpostn(); // Get position
        char *x_setpostn(); // Set position
        char *x_destroy(); // Destroy
    } x_privates; // Private operations
    struct {
        char *x_inlines(); // Inline buffer
    } x_inlines; // Inline buffer
    struct {
        char *x_getbytes(); // Get bytes
        char *x_putbytes(); // Put bytes
    } x_getbytes; // Get/put bytes
};
```

Macros for accessing operation's `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in this section "non-filter primitives" (page 195). The operation `x_inlines()` takes two parameters: an XDR *and an unsigned integer, which is a byte count.

The routine returns a pointer to a piece of the streams internal buffer. The caller then uses the buffer segment for any purpose. From the point of view of the stream, the bytes in the buffer segment have been consumed or put.

The routine may return NULL. If it cannot return a buffer segment of the requested size. (The `x_inlines` routine is for cycle squeezers. Use of the resulting buffer is not data portable. Programmers are encouraged not to use this feature).

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream. They return `TRUE` if they are successful and `FALSE` otherwise. The routines have identical parameters (replace `xxx`).

197
The operations x_getlong() and x_putlong() receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The system primitives HTONL() and NTOHL() can be helpful in accomplishing this. The section "XDR standard" (page 201) defines the standard representation of numbers. The higher level XDR implementation assumes that signed long integers contain the same numbers of bits, and that non-negative integers have the same bit representations as unsigned integers.

The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```c
Bool_t
Xxxlong (xdrs, ip)
XDR *xdrs;
Long *ip;
```

Implementers of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

### 8.8.1 XDR standards

This section defines the XDR standards. This standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on an SCO system, but is consumed by a sun workstation, or conversely. Similarly, the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or PASCAL program.

The XDR standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed the hardware that encodes bytes onto various media preserves the meaning of those bytes across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded using the "little Indian" format. Hardware implementation of both sun workstation and SCO platforms adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a variant of C in that it is a data description language, not a programming
language. In a similar way, the Xerox courier standard uses a variant of Mesa as its data description language.

**Basic block size**
The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read from or written to some byte stream such that byte m always precedes byte m+1.

**Integer**
An XDR signed integer is a 32-bit piece of data that encodes an integer in the range [-2147483648, 2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is integer.

**Unsigned integer**
An XDR unsigned integer is a 32-bit piece of data that encodes a nonnegative integer in the range [0, 4294967295]. It is represented by an unsigned binary number whose most and least bytes are 0 and 3, respectively. The data description of unsigned integers is unsigned.

**Enumeration's**
Enumeration's are useful for describing subsets of the integers. Enumeration's have the same representation as integers. The data description of enumerated data is as follows:

```c
Typedef enum {name = value...} type - name;
```

For example, the three-color red, yellow, and blue could be described by an enumerated type:

```c
Typedef enum {RED =2, YELLOW =3, BLUE = 5} colors;
```

**Boolean**
Boolean's are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```c
Typedef enum {FALSE =0, TRUE =1} Boolean;
```

**Hyper integer and hyper unsigned**
The standard also defines 64-bit (8-byte) numbers called hyper integer and hyper unsigned. Their representations are the obvious extensions of the integer and unsigned above. The most and least significant bytes are 0 and 7, respectively.

**Floating point and double precision**
The standard defines the encoding for the floating-point data types float (32 bits or 4 bytes) and double (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single-and-double precision floating point numbers. (See the IEEE floating-point standard for more information). The standard encodes the following three fields, which describe the floating point number.

S the sign of number. Value 0 and 1 represents positive and negative, respectively.

E the exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.

F the fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating-point number is described by:

\((-1)^S \times 2^{E-bias} \times 1.F\)

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning and most significant bit offsets of S, E, and F are 0, 1, and 9, respectively.

Doubles have the analogues extensions. The beginning and most significant bit offsets of S, E, and F are 0, 1 and 12 respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow) and denormalized numbers (underflow). Under IEEE specifications, the "NAN" (not a number) is system-dependent and should not be used.

**Standard opaque data**

At times, fixed sizes uninterpreted data needs to be passed among machines. This data is called opaque and is described as:

```
Typedef opaque type-name [N];
Opaque name [N];
```

Where [N] is the (static) number of bytes necessary to contain the opaque data. If N is not a multiple of four, the Nbytes are followed by enough (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

**Counted byte strings**
The standard defines a string of n (numbered 0 through n-1) bytes to be the number n encoded as unsigned, and following by the n bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
Typedef string type -name <N>;
Typedef string type -name < >;
String name <N>;
String name < >;
```

The data description languages use the angle brackets (<and >) to denote anything that is of verifying length, as opposed to square brackets to denote fixed-length sequences of data.

The constant N denotes an upper bound of the number of bytes that a string may contain. If N is not specified, it is assumed to be 232-1, the maximum length. The constant N would be found in a protocol specification. For example, a filling may state that a file name can be no longer than 14 bytes, such as:

```
String filename <14>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

**Fixed arrays**

The data description for fixed size arrays of homogenous elements is as follows:

```
Typedef elementtype type- name [n]
Elementtype name [n];
```

Fixed size arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1.

**Counted arrays**

Counted arrays provide the ability to encode variable-length arrays of homogenous elements. The array is encoded as the element count n (an unsigned integer), followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n-1. The data description for counted arrays is similar to that of the counted strings:

```
Typedef elementtype type - name <N>;
Typedef elementtype type - name <>;
```
Again, the constant \( N \) specifies the maximum acceptable element count of an array; if \( N \) is not specified, it is assumed to be \( 2^{32}-1 \).

**Structures**

The data description for structures is very similar to that of standard C:

```c
typedef struct {
    Component -type component - name;
    ...
} type name;
```

The components of the structure are encoded in the order of their declaration in the structure.

**Standard discriminated unions**

A discriminated unions is a type composed of a discriminate followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```c
typedef union switch (discriminant-type) {
    Discriminant-value: arm type;
    ...
    default : default-arm-type;
} type-name
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take place on unspecified discriminant values. Most specifications neither need nor use default arms.

**Advanced topics -linked lists**

This section describes how to pass data structures by using linked lists of arbitrary lengths. Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language:

```c
Long g_assets;
```
Long g_liabilities;
);
bool_t
Xdr_gnumbers (xdrs, gp)
    XDR *xdrs;
    Struct gnumbers *gp;
{
    if (xdr_long (xdrs, & (gp-> g_assets)))
        return (xdr_long (xdrs, & (gp-> g_liabilities)));
    return (FALSE);
}

Now assume that you want to implement a linked list or such information. You could construct a data structure as follows:

#typedef struct gnode {
    Struct gnumbers gn_numbers;
    Struct gnode *nxt;
};

typedef struct gnode *gnumbers_list;

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly, the nxt field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the nxt field is also the address where it continues. The oink addresses carry no useful information when the object type is serialized.

The xdr data description of this linked list is described by the recursive type declaration of gnumbers_list:

Struct gnumbers {
    Unsigned g_assets;
    Unsigned g_liabilities;
};
typedef union switch (boolean) {
    case TRUE: struct {
        struct gnumbers current_element;
        gnumbers_list rest_of_list;
    };
    case FALSE: struct {};
} gnumbers_list;

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a gnumbers structure and (recursively) by a
gnumbers_list (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (through the nxt field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de) serialize a linked list of entries can be found in the XDR description of the pointer-less data. The set consists of the mutually recursive routine xdr_gnumbers_list, xdr_wrap_list, and xdr_gnode.

```c
bool_t
xdr_gnode (xdrs, gp)
    xdr* xdrs;
    struct gnode* gp;
{
    return (xdr_gnumbers (xdrs, &gp->gn-numbers) &&
            xdr_gnumbers_list (xdrs, &gp->nxt));
}
bool_t
xdr_wrap_list (xdrs, glp)
    XDR*xdrs;
    Gnumbers_list*/glp;
{
    return (xdr_reference (xdrs, glp, sizeof (struct gnode),
                        xdr_gnode));
}
struct xdr_discrim choose[2] ={
    /* Called if another node needs (de) serializing */
    {TRUE, xdr_wrap_list},
    /* Called when there are no more nodes to be (De) serialized */
    {FALSE, xdr_void}
}
bool_t
xdr_gnumbers_list (xdrs, glp)
    xdr*g xdrs;
    gnumbers_list* glp;
{
    bool_t more_data;

    more_data = (*glp != (gnumbers_list)NULL);
    return (xdr_union (xdrs, &more_data, glp, choices,
                        NULL));
}
```

The entry routine is xdr_gnumbers_list(); its job is to translate between the Boolean value more_data and the list pointer values. If there is no more data, the
xdr_union () primitive calls xdr_void () and the recursion is terminated. Otherwise, xdr_union () calls xdr_wrap_list (), whose job is to dereference the list pointers. The xdr_gnnode () routine actually (de) serializes data of the current node of the linked list, and recursively calls xdr_gnumbers_list () to handle the remainder of the list. Readers should convince themselves that these routines function correctly in all three directions (XDR_ENCODE, XDR_DECODE, XDR_FREE) for linked lists of any length (including zero). Note that the Boolean more_data is always initialized, but in the XDR_DECODE case it is overwritten by an externally generated value. Also note that the value of the bool_t is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de) serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as xdr_reference, xdr_union, and xdr_void).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below:

```c
Bool_t
Xdr_gnumbers_list (xdrs, glp);
    XDR *xdrs;
    Gnumbers_list *glp;
{
    Bool_t more_data;

    while {TRUE} {
    more_data = { glp != (gnumbers_list) NULL};
    if (! Xdr_bool (xdrs, &more_data))
        return (FALSE);
    if (! More_data)
        return (TRUE); /* we are done */
    if (! Xdr_reference (xdrs, glp, sizeof (struct gnnode);
                  xdr_gnumbers))
        return (FALSE);
    glp =& ((*glp) -• Nxt);
}
```

The claim is that is one routine is easier to code and understand than the three primitives routines above. The parameter glp is treated as the address of the pointer to the head of the remainder of the list to be (de) serialized. Thus, glp is set to the address of the current node's nxt field at the end of the while loop. The discriminated union is implemented in line, the variable more_data has the same
use in this routine as in the routines above. Its value is recomputed and again (de) serializes in each iteration of the loop. Since *glp is a pointer to a node, the pointer is dereferenced using xdr_reference(). Note that the third parameter is truly the size of a node (data value plus nxt pointer), while xdr_gnumbers() only (de) serializes the data values. This optimization works only because the nxt routine in the XDR_FREE case, in that xdr_reference() will free the node *glp. Upon return, the assignment glp = & ((glp)-nxt) cannot be guaranteed to work, since *glp is no longer a legitimate node. The following code works in all cases. The hard part is to avoid dereferencing a pointer that has not been initialized or that has been freed.

This is the first program in this document that actually inspects the direction of the operation (Xdrs-> x_op). The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

```c
Bool_t
Xdr_gnumbers_list (xdrs, glp)
    XDR *xdrs;
    Gnumbers_list *glp;
{
    bool_t more_data
    bool_t freeing;
    Gnumbers_list next; / the next value of glp */
    freeing = (xdrs->x_op == XDR_FREE);
    While (TRUE) {
        More_data = (*glp != (gnumbers_list)NULL);
        If (! Xdr_bool(xdrs, &more_data))
            Return (FALSE);
        If (! More_data)
            Return (TRUE); /* we are done */
        If (Freeing)
            Next = & ((*glp) -> nxt);
        If (! Xdr_reference (xdrs, glp, sizeof(struct gnnode),
            Xdr_gnumbers ))
            Return (FALSE);
        Glp = (Freeing) ? next : & ((*glp) -> nxt);  
    }
}
```

8.9 Application Program Interfaces

Application Program Interfaces which was first introduced in 1986 are the programs that allow user processes to access transport providers in a almost
transport independent fashion. In this chapter we will discuss about the two application program interfaces (APIs) XTI and TLI.

The name of the library to be searched when compiling and linking a program that uses TLI is NSL (Network Services Library)

Applications written to use TLI can be ported relatively easily to XTI. The following points are noted when applications are ported.

- The t_error event is not defined in XTI. Consequently, t_look never returns this event as its result value.
- XTI defines the events T_GODATA and T_GOEXDATA to assist in managing flow control.
- The flags O_RDONLY and O_WRONLY are not defined in XTI. You must change them to O_RDWR.

8.10 Transport Addresses

A network consists of multiple computers connected to each other in a way that allows one computer to communicate with any of the other computers on the network. A network may be connected to other networks either directly or indirectly with the help of software or Hardware.

The transport provider running on a host computer must be able to identify both that host computer and the network to which the host computer is attached. This information is referred as the "Network Address" and the "host id". In Network Management we already discussed that each kind of transport provider has its own schemes.

Being able to identify a particular host and the network to which it is attached is not enough, however. With a multi tasking operating system such as UNIX, many processes (whether clients, servers, or both) can be running on the same host simultaneously. As a result, the transport provider must be able to uniquely identify every process on the host that communicates with the transport provider. Each kind of transport provider defines its own identifier (the "local process id").

Once the process (whether client or server) has opened a transport endpoint with a specific transport provider, it needs to establish the network address and "host id" of the machine it is running on, as well as the local process id by which it will be known. A process associates this information with the transport endpoint by calling the t_bind function.

The process can neither select the network address, host id, and local process-id to be used, or it can allow the transport provider to select them. Even if the process makes the selection, the transport provider can override it. In either case, the process can, if it wants, find out the network address, host id, and local
process id selected by the transport provider. It does this by providing the appropriate argument to \texttt{t\_blind}.

Generally, servers specify their local process-id and clients do not. This is because clients are written to contact a particular server using the predetermined local process id of that server. It is crucial, therefore, that the server use the predetermined local process id and no other. On the other hand, when the client connects to a server, the local process id of the client is passed to the server accepts the connection. The actual value of the local process id being used by the client is of no importance.

8.11 Types of Services

Basically the transport provider offers two types of services i.e. connection less and connection oriented services. In the earlier days when Networking is in the initial stage the connection oriented services are very common but as the net technology evolved the connectionless services are preferred.

8.11.1 Connection- oriented services

In “connection-oriented” service, a client and a server establish a communication path over which they can send and receive data. This kind of service is useful when a client and server expect to have an extended dialogue with each other. The characteristics of connection-oriented service is that the communication is sequenced, reliable, and error free.

A client and a server must cooperate to set up a connection. The server prepares to receive an incoming connection request by calling \texttt{t\_listen}. The client then calls \texttt{t\_connect} to request a connection with that server. Finally, the server accepts the connection request by calling \texttt{t\_accept}.

8.11.2 Connectionless services

In “connectionless” service, the client and the server exchange individual messages. It is useful when the service or computation provided by the server requires very little interaction between the server and the client.

Typically, the client and server begin by calling \texttt{t\_open} and \texttt{t\_blind}. The client then calls \texttt{t\_alloc} to allocate space for the message to be sent, fills in the data area with the message, then sends the message to the server by calling \texttt{t\_sndudata} (send unit data). Meanwhile, the server awaits the incoming message by calling \texttt{t\_rcvudata} (receive unit data). After the client sends its message, it calls \texttt{t\_rcvudata} and waits for the server to respond. When the client’s messages arrives at the server, the server calls to \texttt{t\_rcvudata} completes. The server can than examine the message, perform whatever computation it was designed to do,
and send its reply by first calling t_alloc, filling the data area with its reply, then sending it off with t_sndudata.

The client and server continue to exchange messages this way until they are finished.

As compared to the connection-oriented service, a connectionless service does not guarantee delivery of messages, does not ensure that messages arrive at their destination in the same order in which they were sent, and does not ensure that data arrives error-free.

8.12 Synchronous and Asynchronous Operation

In synchronous mode, a function call does not return until the operation can be completed. For example, if a process calls t_rcv in synchronous mode and no data is available, the call blocks until data arrives at the transport mode and no data is available, the call blocks until data arrives at the transport endpoint. On the other hand, if the process calls t_rcv in asynchronous mode and no data is available, the call returns immediately with a value of -1 and the global variable t_errno is set TNODATA. It is then up to the process to decide when and how to try again and call t_rcv later.

The following functions that is used for either synchronously or asynchronously are:

- t_connect
- t_rcvconnect
- t_listen
- t_rcv
- t_rcvudata
- t_snd
- t_sndudata

A process usually indicates whether these functions will operate synchronously or asynchronously be setting the oflag argument to t_open to o_NONBLOCK.

8.13 Flow of Transmitted Data

One of the basic characteristics of both connection-oriented and connectionless service is flow control. This means that a process cannot send data through a transport provider already has all the data it can currently handle.

If a transport endpoint is operating in synchronous mode and flow control is in affect, a call to t_snd or t_sndudata will block until the transport provider has
freed up enough internal storage (buffers) to accept more data. The transport provider frees buffers by transmitting data to the destination transport endpoint.

If a transport endpoint is operating in asynchronous mode and flow control is in effect, a call to t_snd or t_sndudata returns -1 and the global variable t_errno is set to TFLOW. The process can either call the t_look function to detect when it can send more data (t_look returns either T_GODATA or T_GOEXDATA when flow control has been lifted) or it can use an event management facility to be notified when flow control has been lifted.

When a process reads data from a transport endpoint in connection-oriented mode, the data can appear to be either continuous stream of bytes (stream oriented input) or a sequence of messages with message boundaries preserved (record oriented input).

In the case of a continuous byte stream, the reading process cannot tell how many individuals messages were sent to it, nor where one message stops and the next one begins. If message boundaries are preserved, however, the reading process can detect the end of one message and the start of another. Depending on the transport provider being used, a process reading incoming data on the other side of these two forms.

When a process reads data from a transport endpoint in connectionless mode, it always receives a datagram (message). A datagram is a self-continued unit with a start and an end indicator for the data. When a process sends diagrams (by calling t_sndudata), each datagram is sent individually to the remote process. The receiving process then retrieves each datagram (by calling t_rcvudata) one at a time.

This is different from sending data over a connection. In connection-oriented mode, the sending process calls t_snd to submit data. A process may call t_snd several times before it turns around and waits to receive data from the remote process (it does this by calling t_rcv). At the other end, the receiving process may get all of the data that was sent to it with a single call to t_rcv. Because the receiving end process sees a byte system as it reads data, it has no idea how many times the sending process called t_snd, nor what data was sent with each of those calls.

8.14 Priority of Transmitted Data

Some transport providers support the idea of "expedited data". The expedited data is to be sent immediately to the remote process. Expedited data is useful because transport provider operating in connection oriented mode typically don’t send data in small chunks. To improve efficiency, the transport provider waits until the sending process has written enough data before actually transmitting it over the network. This widely used technique is called "buffering."
To make sure that a piece of data is not buffered for some unknown period of time, the sending process can mark it as “expedited”. This is done by calling \texttt{t_snd} and passing in the value \texttt{T_EXPIDETED} as the flag argument. In this case, the transport provider transmits the expedited data ahead of all data currently being buffered without delay.

8.14.1 Problems with Expedited Data

The following are the problems are observed with expedited data:

1. All transport providers do not support transfer of Expedited data.
2. The way in which expedited data actually works on transport providers that do support it (that is, the semantics of expedited data) is not exactly the same for all transports providers.

8.15 Different states of Transport provider

The following states are observed when XTI are used as a Transport provider:

\textbf{T\_UNINIT} \hspace{1cm} This is the “uninitialized” state. In this stage no transport endpoint exists. The only function that can be called from this state (without generating an error) is \texttt{t\_open}.

\textbf{T\_UNBND} \hspace{1cm} This is the “unbound” state. This state results for a successful call to \texttt{t\_open}. In this state a transport endpoint exists, but no network address, host id, and local process has been associated with it.

\textbf{T\_IDLE} \hspace{1cm} This is the “idle” state. This state results from a successful call to \texttt{t\_blind}. In this state a transport endpoint exists, and a network address, host id, and local process id have been associated with it. However, no connection has been established between this transport endpoint and a transport endpoint on another host.

\textbf{T\_OUTCON} \hspace{1cm} This is the “outgoing connection request” state. In this state the transport provider has sent an outgoing connection request to another transport endpoint in response to a clients call to \texttt{t\_connect}.
T_INCON  This is the “incoming connection request” state. An incoming connection request is pending. In this state the transport provider has received a connection request from client.

T_DATAxFER  This is the “data transfer” state. This state is defined for connection oriented service only. The transport provider must be in this state before a process (client or server) can send data over a connection. On the client, this state results from completion of a successful call to t_connect or t_rcvconnect. On the server, this state results from completion of a successful call to t_accept.

T_OUTREL  This is the “outgoing orderly release” state. This state is defined for connection-oriented service only. In this state, the process is waiting to receive an orderly release indication from the remote process. The transport provider enters this state after a call to t_sndrel.

T_INREL  This is the “incoming orderly release” state. This state is defined for connection-oriented service only. In this state, the process is waiting to send an orderly request. The transport provider enters this state after a call to t_rcvrel.

8.16 Errors in execution

Most of the functions return a value of −1 if they encounter an error. In that case the global variable t_errno is set to one of errors defined for the function.

8.17 Options Management

XTI (and TLI) allow a process to negotiable various options with the transport provider by calling t_optmgmt. These options can affect how the transport provider operates. Some of the options require negotiations with the remote process, because both the client and the server must agree on the value of the option. The option available depends on the kinds of transport provider, and can even vary between different implementation of the same kind of transport provider.

8.18 Event management

XTI allow the process to manage multiple transport endpoints in a fully asynchronous manner using an event – driven design. In this kind of design, a process waits for any one of the several events to occur. When an event occurs, the process discovers which event it is, then transfers control to the code written to handle that event. This code can be executed by the process that waited for the
event or the process can fork (S) a child process execute the code. While the child process in executing the parent process can either waits (S) for the child to complete, or it can proceed to process the next event.

8.19 Issues in Transport providers

Although XTI is a transport-independent programming interface, some areas that a network program must confront lie outside the scope of XTI. Two such areas are transport addresses and options management. Address formats and available options vary from transport provider to transport provider. While XTI provides a generic mechanism for assigning and retrieving address and options information, it is up to the software developer to code the specific details appropriate to each transport provider. These details will have to be recorded when the program is ported to use a different transport provider.

Similarly, each transport provider protocol structures stores internal information differently.

Each transport provider can also define its own mechanism to allow programs to refer to hosts-id's or local process-id's with symbolic names, rather than numbers. The defined mechanism converts the symbolic name into the appropriate numeric value.

Pseudo-code for Client XTI(given in ANNEX 1)
Pseudo-code for Server XTI(given in ANNEX 1)
Implementation of TLI Client over TCP/IP(given in ANNEX 1)
Implementation of TLI Server over TCP/IP(given in ANNEX 1)