Automated Development of Distributed Applications

The complications of layering in the OSI model come to a head in the Session Layer and a number of recent developments have 'bypassed' many of the OSI layers.

These include, and have been motivated by :

- Speed,
- Distributed and replicated file systems,
- Remote process invocation and control,
- Network-aware programming languages, such as Java, and
- Increased need for distributed security.

The Remote Procedure Call (RPC) Paradigm

The *remote procedure call* (RPC) paradigm [BJ Nelson, 1981] and [AD Birrell and BJ Nelson, 1984] is based on the observation that procedure calls are a well understood mechanism for control transfer.

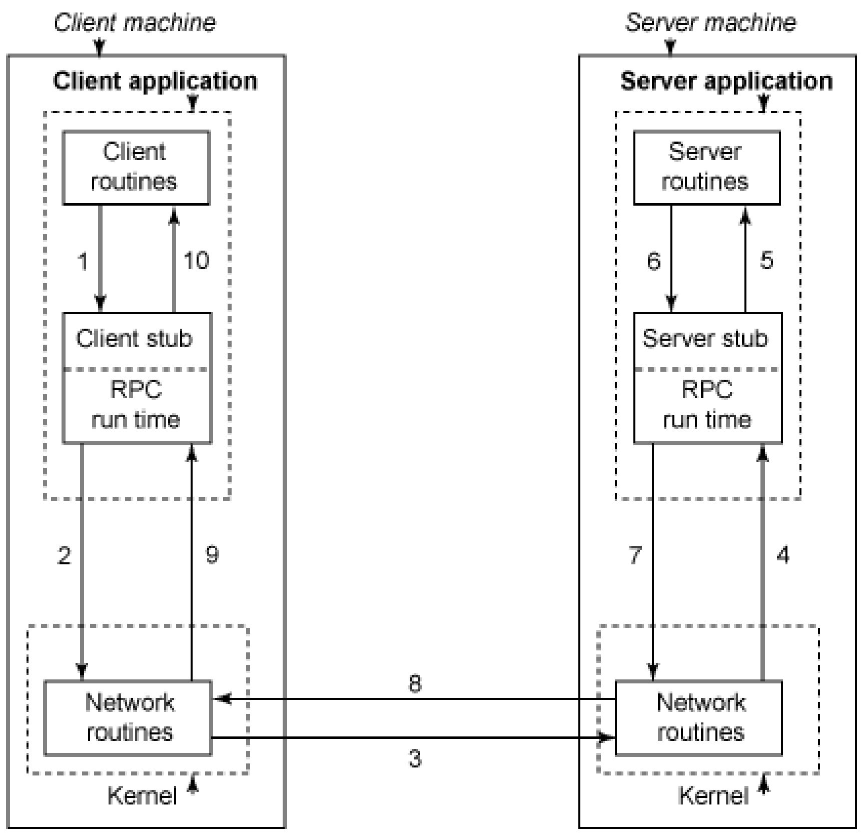
The proposal is that procedure calls may be consistently extended to access remote environments (other machines).

When a remote procedure call is invoked :

- The calling environment is suspended,
- Any parameters are passed *(marshalled)* across the network to the remote environment,
- The required procedure is executed in the remote environment, and
- Results are marshalled back to the caller and its execution resumes.

See: Implementing Remote Procedure Calls, Birrell, Andrew D. and Nelson, Bruce Jay, in ACM Trans. Comput. Systems, 2(1), pp39-59, February 1984.

The RPC Execution Order



1. The client calls a local procedure termed the *client stub*.

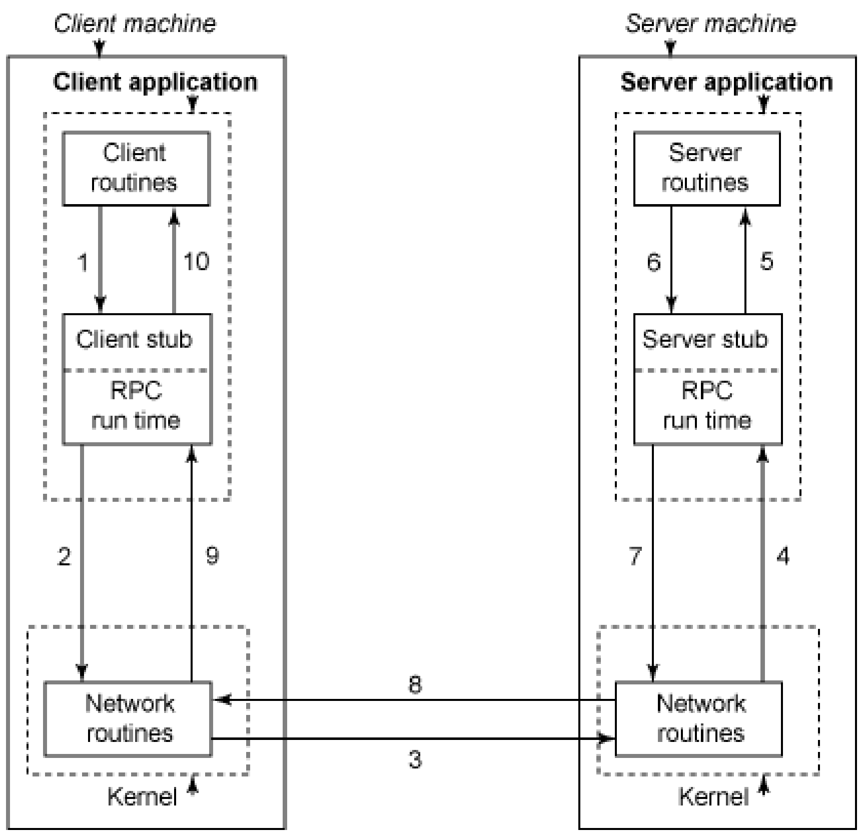
   It appears to the client that the stub is the actual server procedure that it wants to call.

   The purpose of the stub is to package up the arguments to the remote procedure, possibly put them in some standard form and then to build one or more network messages (*marshalling*).

2. The network messages are sent to the local kernel using a system call.

3. The network messages are sent to the remote kernel using either a connection based or connectionless protocol.

The RPC Execution Order, *continued*



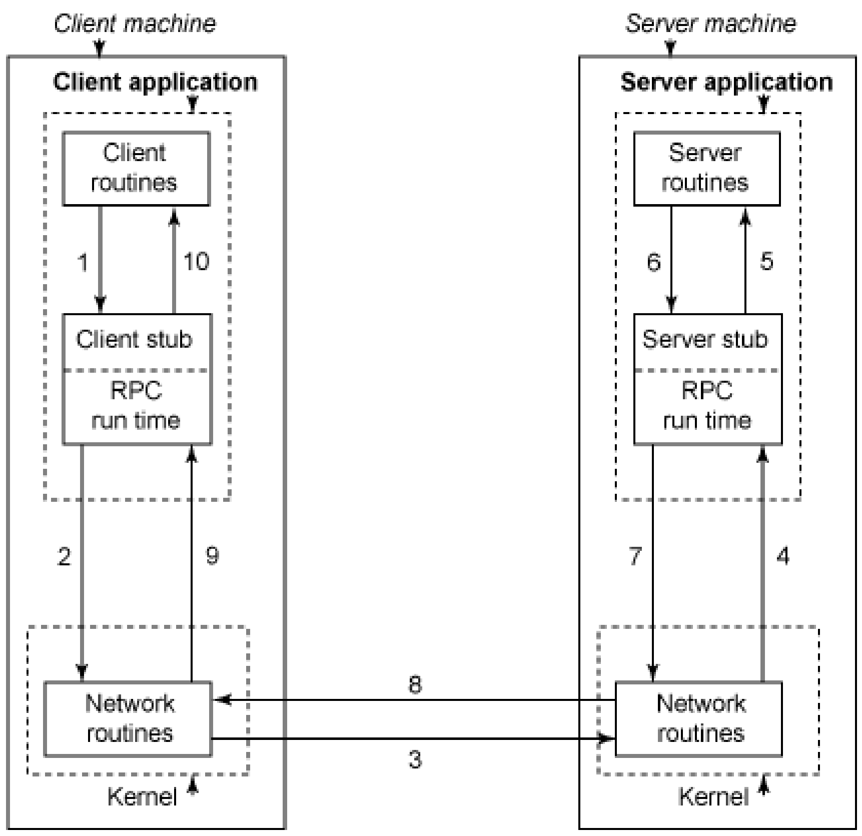
4. The *server stub* has been waiting on any client's request.

   It unmarshals the arguments from the network messages and possibly converts them to its own (architecture's) format.

5. The server stub executes a local procedure call to invoke the actual server function.

6. When the server procedure is finished it returns (normally) to the server stub, returning any required arguments.

The RPC Execution Order, *continued*



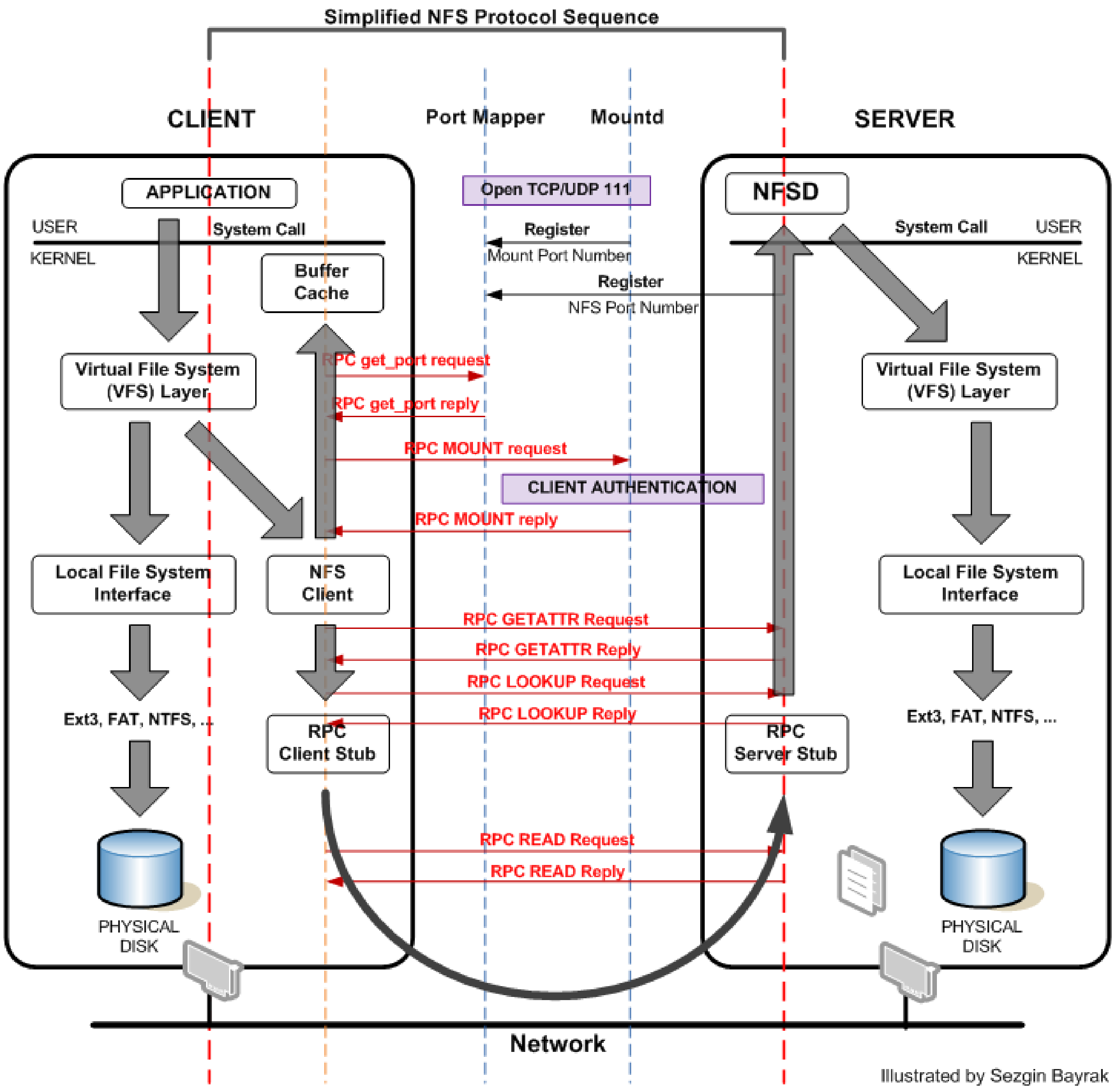
7. The server stub converts the return values, if necessary, and builds one or more network messages.
8. These messages traverse the network.
9. The client stub reads the replies from the local kernel (it has blocked all this time).
10. After possibly converting the return values the client stub returns to the calling procedure; control flow is again in the client's code.

An Example of Transparent Access

Consider an application requiring access to a file available on another machine.

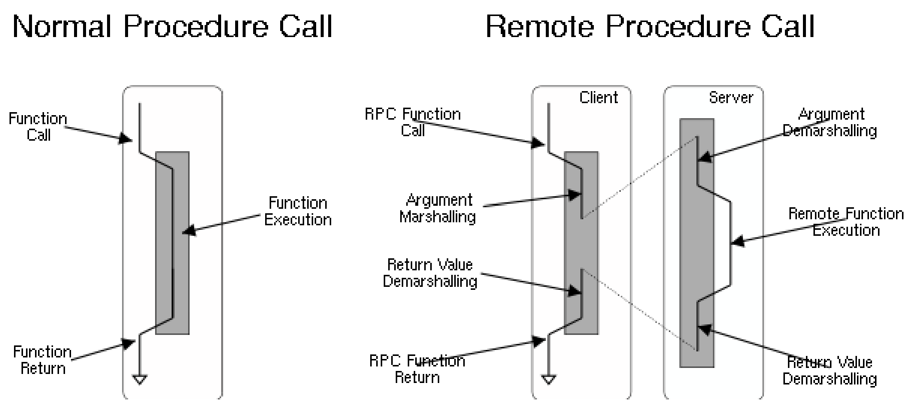
```
fd = open( "/home/uniwa/staff9/staff/00012349/linux/src/example.c", O_RDONLY, 0);
```

Here, the local operating system kernel recognized that the *mount point*, `/home/uniwa/staff9`, refers to a file system on a remote machine and a series of RPC requests are made to access the remote file.



Passing parameters to remote procedures

Significant problems are introduced with parameter passing:



- Passing simple data types (*by value*) such as integers, floating point numbers and strings (character arrays) is easy. The client-stub simply places *copies* of them in a calling packet.
- At worst, data types may need format conversion between machines (performed by the Presentation Layer). There exists a well defined data interchange standard - the External Data Representation (XDR) for transferring data between heterogeneous environments (in [RFC1014]).
- To get an appreciation of the complexity of XDR, read the manual entry for `xdr` or the header file (on Linux).
- Trouble arises when *reference* parameters must be passed. When local procedures are invoked a reference parameter, such as a *pointer* may be passed and followed by the procedure (they have the same address space).

This is not true of RPCs!

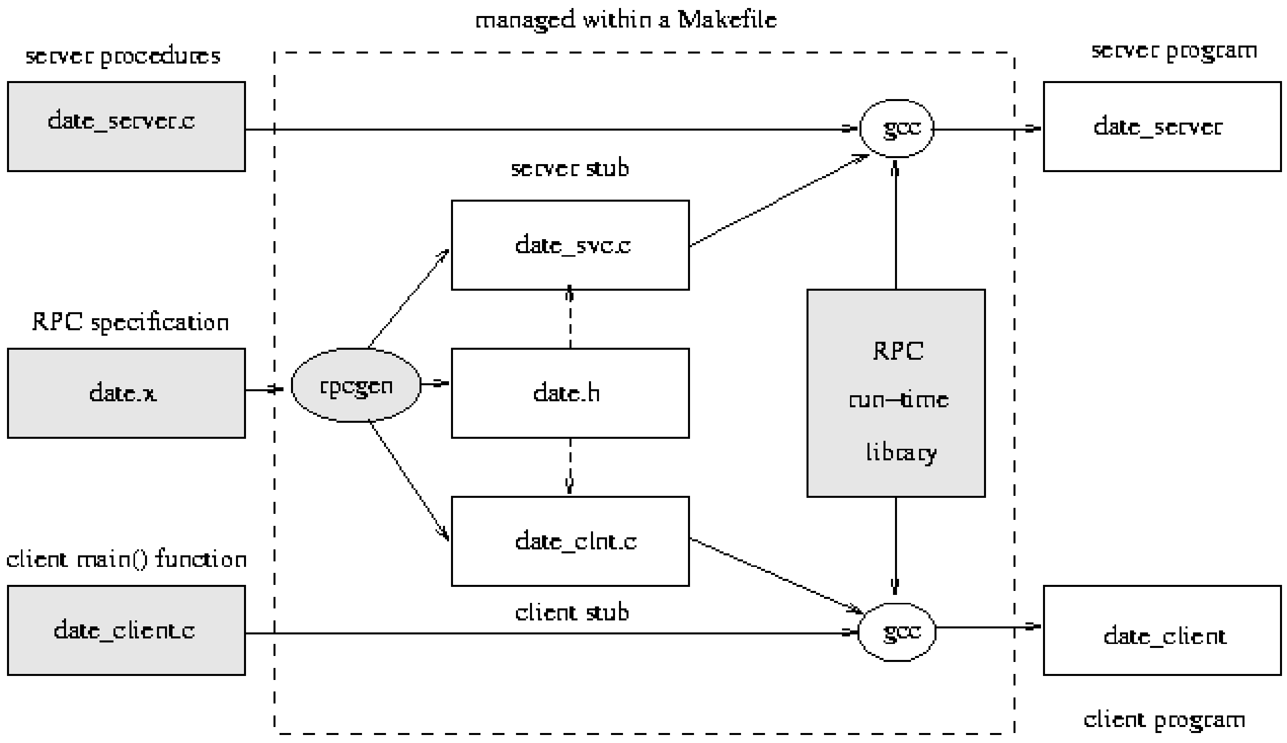
SUN Microsystem's RPC Compiler - *rpcgen*

Many operating systems provide RPC within their kernels and as a standard library of routines.

One such example is SUN's implementation of RPC (in [RFC1057]). The RPC package consists of `rpcgen`, a compiler for creating remote procedure call server and client stubs, the XDR for encoding data into a portable manner between different architectures and a runtime library (provided in C's standard library `libc`).

For example:

```
program DATE_PROG {
   version DATE_VERS {
      long     BIN_DATE(void) = 1;    /* proc #1 */
      string   STR_DATE(long) = 2;    /* proc #2 */
   } = 1;                             /* version 1 */
} = 0x37621;                          /* program number */
```



Complete example in rpc_example.zip

RPC Client-side Code in C

```
//  date_client.c - client program for remote date service.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <rpc/rpc.h>   // standard RPC include file
#include <rpc/auth.h>

#include  "date.h"          // automatically generated by rpcgen

typedef unsigned int    rpc_uint;

int main(int argc, char *argv[])
{
    CLIENT  *handle;        // RPC handle, used in all RPC transactions
    char    *remote_host;

    long    *lresult;       // return value from bin_date_1()
    char    **sresult;      // return value from str_date_1()

    if(argc != 2) {
        fprintf(stderr, "Usage: %s hostname\n", argv[0]);
        exit(1);
    }
    remote_host = argv[1];

//  CREATE THE CLIENT "HANDLE" TO BE USED IN ALL RPC TRANSACTIONS.
    if((handle = clnt_create(remote_host, DATE_PROG ,DATE_VERS, "udp")) == NULL) {
        clnt_pcreateerror(remote_host); // no connection with server.
        exit(2);
    }

//  FIRST CALL THE REMOTE PROCEDURE "bin_date"
    if((lresult = bin_date_1(NULL, handle)) == NULL) {
        clnt_perror(handle, remote_host);
        exit(3);
    }
    printf("time on host %s = %ld\n", remote_host, *lresult);

//  NOW CALL THE REMOTE PROCEDURE "str_date"
    if( (sresult = str_date_1(lresult, handle)) == NULL) {
        clnt_perror(handle, remote_host);
        exit(4);
    }
    printf("time on host %s = %s", remote_host, *sresult);

//  CLOSE OUR CONNECTION WITH THE RPC SERVER
    clnt_destroy(handle);

    return 0;
}
```

RPC Server-side Code in C

```
//  date_server.c - remote procedures; called by server stub

#include  <time.h>
#include  <rpc/rpc.h>           // standard RPC include file
#include  "date.h"              // this file is generated by rpcgen

//  RETURN THE DATE AND TIME AS AN INTEGER
long *bin_date_1_svc(void *unused1, struct svc_req *unused2)
{
    static long timeval;        // must be declared as a static

    timeval = time(NULL);

    return (&timeval);
}

//  RECEIVE AN INTEGER TIME VALUE AND RETURN A HUMAN READABLE STRING
char **str_date_1_svc(long *bintime, struct svc_req *unused)
{
    static char *ptr;           // must be declared as a static

    ptr = ctime(bintime);       // convert to local time

    return (&ptr);              // return the address of pointer
}
```

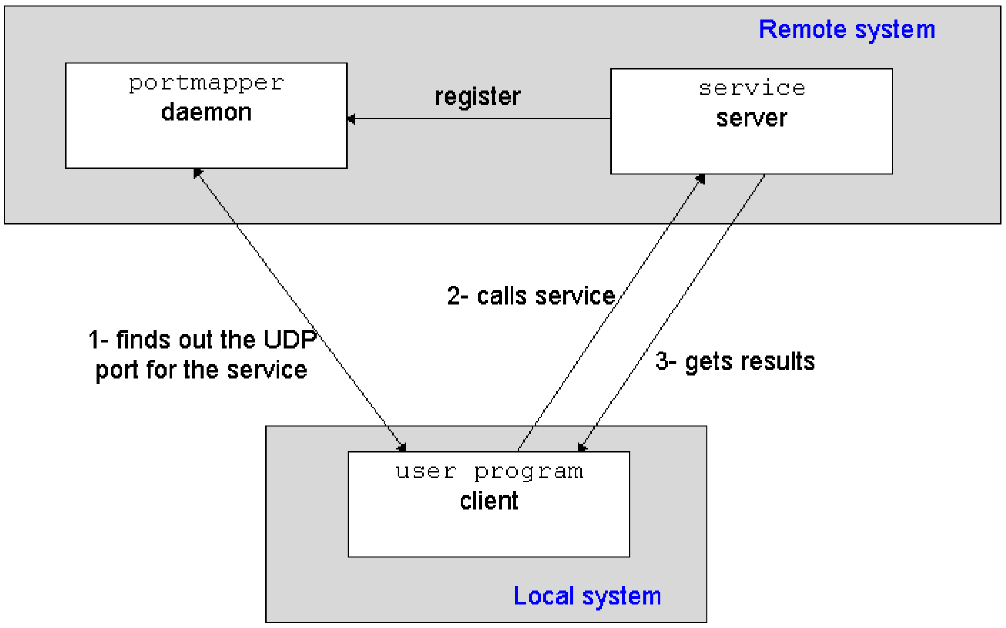
Naming and Interface Binding

How does the client-stub know who to call?

Most operating systems now supporting RPCs use a replicated database used to store server addresses.

When a server restarts (boots) it informs the database that it is alive and passes it :

- the program's program number,
- the program's version number, and
- its port number (on that machine).

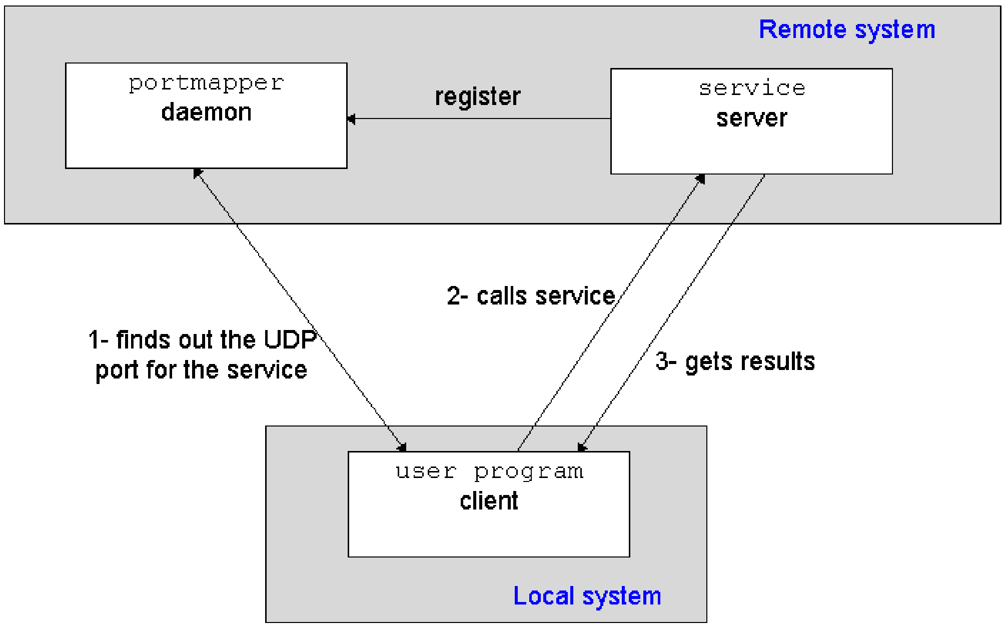


Thereafter, the first time a client-stub needs to locate a remote procedure it first asks the database server (*the portmapper*).

The server maps the procedure's name to the network address.

This process is termed *binding*.

Locating and calling the server



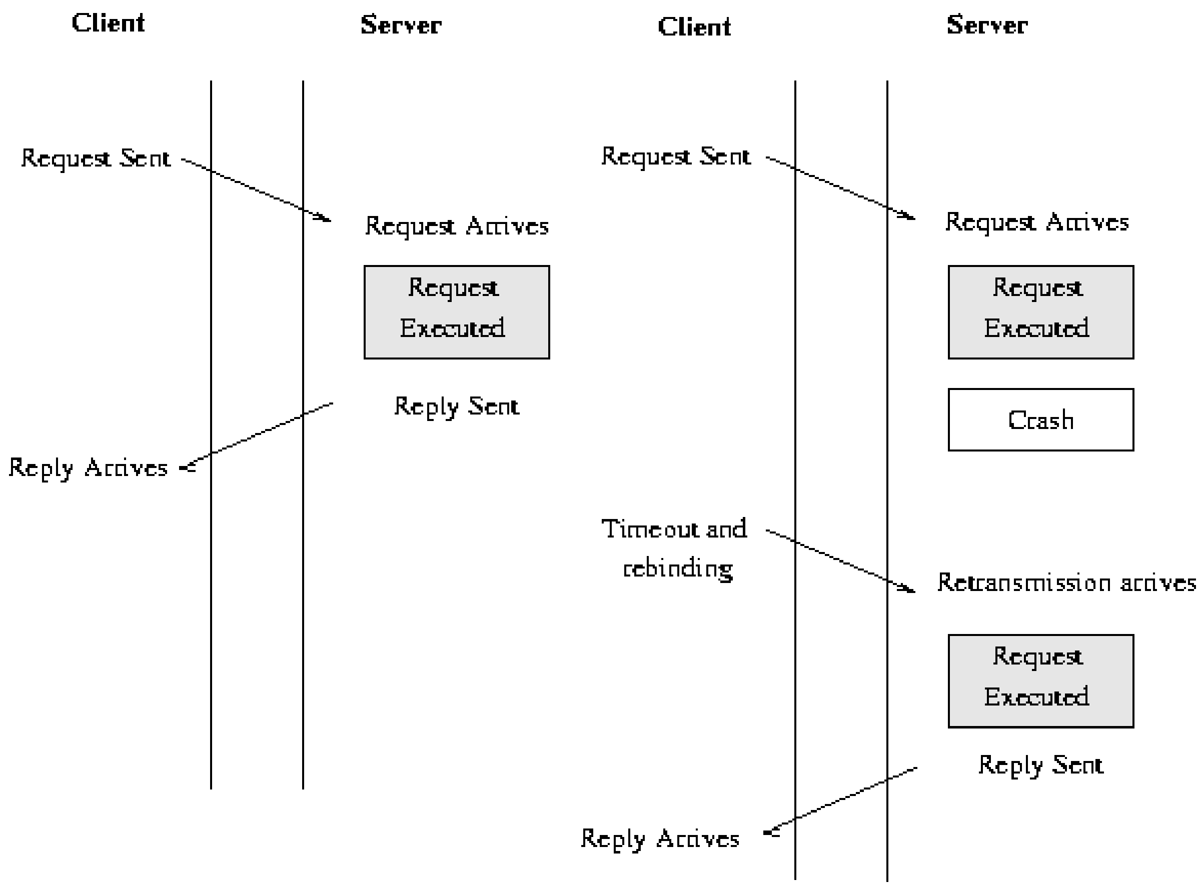
- When we start the server program on the remote machine it creates a UDP socket and binds any local port to that socket. The function `svc_register` in the RPC library is called to register the server with the *portmapper* process. The portmapper keeps track of each server's program number, version number and port number.
- We start our client program which calls `clnt_create`. This function contacts the portmapper on the remote system to determine the UDP port number.
- Our client calls the `bin_date_1` function (the client stub). The stub sends the 'call' to the server stub using a UDP datagram. An integer is returned as the single result.
- Our client calls the `str_date_1` function (the client stub). The stub sends the single parameter to the server stub using a UDP datagram. The string result is returned in the parameter array.

Semantics of Remote Procedure Calls

Ideally, Remote Procedure Calls look like local procedure calls and the application programs may be unaware of the existence/need for the network. Like everything else, they suffer from network crashes, lost messages and delays.

Consider what happens when a server crashes. The client stub may :

- Block and await the reply (which will never come).
- Time-out and report a failure, or exception, to the client.
- Time-out and retransmit the request.



Should clients *re-issue* their requests in the event of a failure? Moreover, should the client's application (manually) or the client's stubs (automatically) re-issue a request?

It is important to understand whether a package supports *at-most-once* or *at-least-once* semantics. Remote operations, which may be repeated without consequence, are termed *idempotent operations*.

The External Data Representation

The External Data Representation (XDR) is a standard for the description and encoding of data [RFC1014].

XDR was designed specifically to provide the marshalling and unmarshalling operations for Sun's implementation of RPC.

All data which is transferred in RPC is translated using XDR.

XDR is also useful in situations which do not use RPC, or even the network, as it allows one to read and write arbitrary C data structures in a consistent and well-defined manner.

e.g. We can use XDR to save a program's state (i.e. data structures) to a file so that when the program is restarted it can resume execution where it left off.

XDR fits into the ISO presentation layer, and is roughly analogous to ISO's Abstract Syntax Notation.1.

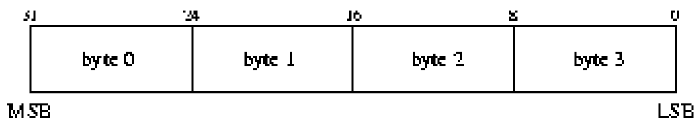
The major difference between the two is that ASN.1 explicitly sends typing information along with the data while in XDR this information is implied.

The Differences in Data Representation

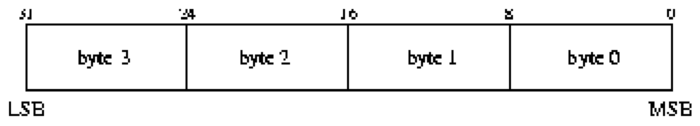
Machines of different architectures represent data differently internally.

Simple example: integers

On Sun-SPARC or Motorola PowerPC architectures integers are stored as



while on Intel x86 processors, integers are stored as



So the integer `1` on a SPARC or PowerPC would be interpreted as the integer`16777216` (2^{24}) on the Intel.

Other problems occur with respect to alignment and pointers:

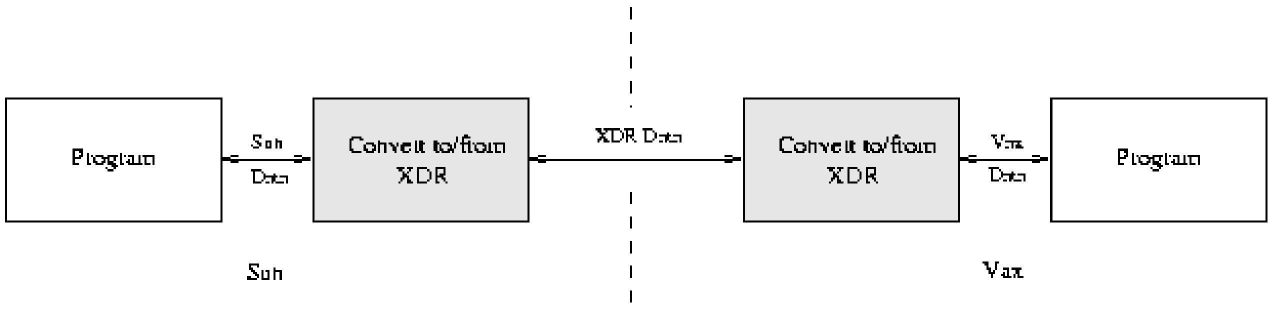
- Different alignment schemes will cause the data in structures to be stored differently.
- Pointers have no meaning outside the program where they are used.

The XDR Approach

XDR takes a *canonical* approach to data communication: it defines a standard XDR representation for data and makes its clients use it.

If a program wishes to use XDR to transmit data it must firstly translate its internal representation of the data to the XDR representation.

A program receiving XDR information performs the opposite mapping: it converts the incoming XDR data to its own representation



The advent of a new machine/language has no impact on existing users: the new machine is 'taught' to convert between XDR and its own representation and can thereafter communicate with all other XDR users.

The XDR Data Representation

- The representation used by XDR is defined fully in RFC1014.
- XDR assumes that bytes (octets) are portable between architectures.
- The representation of all objects requires a multiple of 4 bytes, numbered *0* to *n-1*.
- The bytes are read from and written to streams such that byte *m* precedes byte *m+1*.
- If the object being represented is not a multiple of *4* bytes in length then the*n* bytes are followed by enough *0* bytes to make the total byte count a multiple of*4*.
- An unfortunate consequence of this is that sending a single character will involve a 75% waste of bandwidth (!).
- The standard, however, defines representations for arrays of characters to minimise this wastage in general use.

The XDR Representation, *continued*

XDR defines the representation of simple types, and the representation to be used when combining these types to produce more complex types.

The simple types include:

- integers, short and long, signed and unsigned
- floats, single and double precision
- characters, signed and unsigned
- enumerated types and Booleans

These may be combined to produce the complex types:

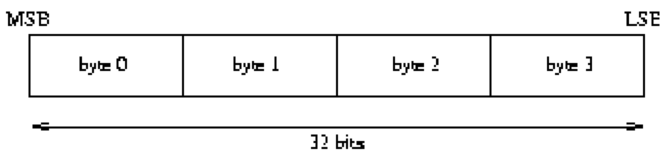
- fixed and variable length arrays
- strings
- structures
- discriminated unions

XDR also allows some special types:

- fixed and variable length opaque data
- the 'void' type

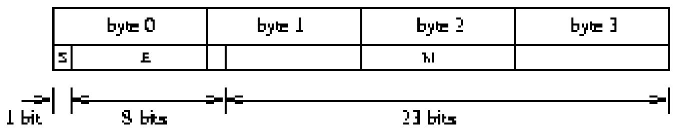
The XDR Representation of some simple types

Integers:

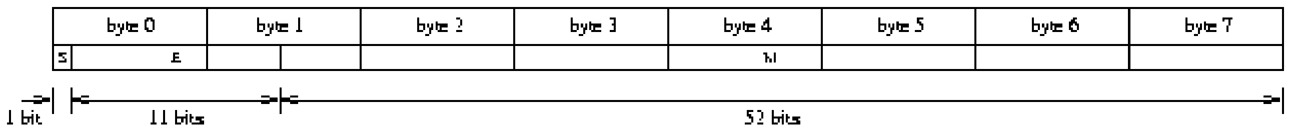


Signed: Two's Complement (MSB is a sign bit).
Unsigned: As above, but all 32 bits store the value.
Short Integers: Stored as though the short had been assigned to a long.

Reals - Single Precision:



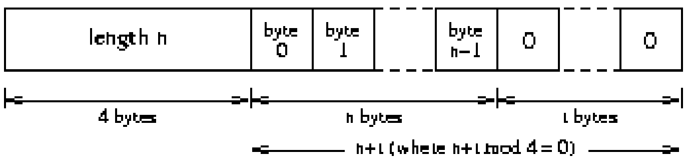
Reals - Double Precision:



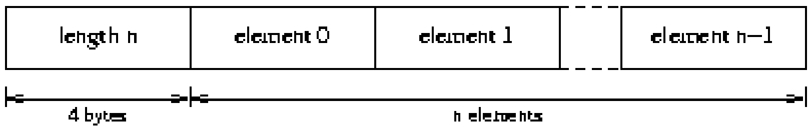
S - Sign Bit (0 -> positive)
E - Exponent
M - Mantissa

The XDR Representation of some complex types

Strings:

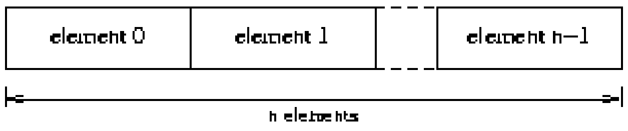


Variable Length Arrays:

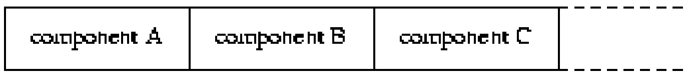


Note that a string is represented more efficiently than an array of characters would be.

Fixed Length Arrays:



Structures:



The SunOS XDR Library

The SunOS library contains functions to convert each of the primitive types to/from their XDR representation.

- Users write their own functions to convert more complex data structures.
- Each XDR conversion routine (including those written by users) takes two parameters: an XDR stream pointer and a pointer to the data to convert.
- An XDR stream is a handle which is used to specify the source/destination of XDR data.
- Ways exist of obtaining XDR streams which are connected to standard input/output (e.g. files opened using `fopen()`), memory (useful for bundling data before sending it off as a datagram), and TCP/IP stream connections.
- When an XDR stream is created the user specifies whether it will be used for encoding (symbolic constant `XDR_ENCODE`) or decoding (symbolic constant `XDR_DECODE`).
- The same XDR conversion routines are used for both encoding and decoding data: the type of the XDR stream tells the conversion routine what to do.

The SunOS XDR Library, *continued*

Example:

The XDR routine to convert an integer is declared as

```
bool    xdr_int(XDR *xdrs, int *ip)
```

If we wish to XDR encode an integer and send it to standard out we would firstly create an XDR stream to encode data:

```
XDR *xdrs;
int i;
        .
        .
xdrstdio_create(xdrs, stdout, XDR_ENCODE);
```

and then call the function

```
i = 23;
if(!xdr_int(xdrs, &i)) {
        error-handling
}
```

To decode reading from standard input we would create the stream using

```
xdrstdio_create(xdrs, stdin, XDR_DECODE);
```

and use the same function call

```
if(!xdr_int(xdrs, &i)) {
        error-handling
}
```

to receive the integer into variable `i`.

Converting complex data structures

Example:

If we wished to XDR encode a structure of the form

```
struct person {
    char    name[50];
    int     age;
    char    sex;
}
```

we could use the following XDR conversion function

```
bool    xdr_person(XDR *xdrs, struct person *p)
{
        if(!xdr_string(xdrs, &p->name, 50))
                return false;
        if(!xdr_int(xdrs, &p->age))
                return false;
        if(!xdr_char(xdrs, &p->sex))
                return false;
        return true;
}
```

If we had an array of person structures, declared as

```
struct person    world[200];
```

we could convert it using

```
xdr_vector(xdrs, world, 200, sizeof(struct person), xdr_person);
```

The final parameter is the name of the function which is to be called to XDR encode each element of the array.

Converting complex data structures, *continued*

Example:

If we wished to XDR encode a pointer to the person structure declared as

```
struct person   *pp;
```

we would convert it using

```
xdr_reference(xdrs, &pp, sizeof(struct person), xdr_person);
```

If the XDR stream indicates an encode operation the function follows the pointer and encodes the data it points to by calling `xdr_person()`.

If the XDR stream indicates a decode operation and `*pp` is `NULL` the routine allocates memory to hold the structure and makes `pp` point to that area.

The routine then decodes the data by calling `xdr_person()` and places the decoded structure into the memory pointed to by `pp`.

A related routine called `xdr_pointer()` exists which more correctly understands `NULL` pointers.

This facility can be used to create functions which encode/decode linked lists and other arbitrarily complex data structures.