Originally published at pp. 138-44 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

Sun RPC

RFC 1831 [Srinivasan 1995a] describes Sun RPC which was designed for client-server communication in the Sun NFS network file system. Sun RPC is sometimes called ONC (Open Network Computing) RPC. Sun RPC is supplied as a part of the various Sun UNIX operating systems and is also available with other NFS installations. Implementers have the choice of using remote procedure calls over either UDP/IP or TCP/IP.

The Sun RPC system provides an interface language called XDR and an interface compiler called *rpcgen* which is intended for use with the C programming language.

Interface definition language ◊ The Sun XDR language which was originally designed for specifying external data representations was extended to become an interface definition language. It may be used to specify a Sun RPC interface which contains a program number and a version number rather than an interface name, together with procedure definitions and supporting type definitions. A procedure definition specifies a procedure signature and a procedure number. As only a single input parameter is allowed, procedures requiring multiple parameters must include them as components of a single structure. The output parameters of a procedure are returned via a single result. The procedure signature consists of the result type, the name of the procedure and the type of the input parameter. The type of both the result and the input parameter may specify either a single value or a structure containing several values.

For example, see the XDR definition in Figure 0.1 of an interface with a pair of procedures for writing and reading files. The program number is 9999 and the version number is 2. The *READ* procedure takes as input parameter a structure with three components specifying: a file identifier, a position in the file and the number of bytes required. Its result is a structure containing the number of bytes returned and the file data. The *WRITE* procedure has no result. The *WRITE* and *READ* procedures are given numbers 1 and 2. The number zero is reserved for a null procedure that is generated automatically and is intended to be used to test whether a server is available.

The interface definition language provides a notation for defining constants, typedefs, structures, enumerated types, unions and programs. Typedefs, structures and enumerated types have the same syntax as in C. The interface compiler *rpcgen* can be used to generate the following from an interface definition:

- client stub procedures;
- server *main* procedure, despatcher and server stub procedures. The despatcher can pass authentication information (user id and group id) to the server procedures;
- XDR marshalling and unmarshalling procedures for use by the despatcher and client and server stub procedures;
- a header file, for example *"FileReadWrite.h"*, containing definitions of common constants and types that may be used in client and server programs. The service procedure signatures are given as C function prototypes. The developer of the service provides implementations of these procedures that conform to the prototypes.

Figure 0.1 Files interface in Sun XDR.

```
/*
 * FileReadWrite service interface definition in file FileReadWrite.x
*/
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};
struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
 version VERSION {
        void WRITE(writeargs)=1;
        Data READ(readargs)=2;
        }=2;
} = 9999;
```

Client program ◊ A client program imports the appropriate service interface and calls the remote procedures, for example, *READ* and *WRITE*. It is supported by the client stub procedures and the marshalling and unmarshalling procedures generated by the interface compiler *rpcgen*.

Sun RPC does not have a network-wide binding service. Instead it provides a local binding service called the *port mapper* which runs on every computer. Each instance of a port mapper records the port in use by each service running locally. Therefore to import an interface, the client must specify the hostname of the server as well as the program number and version number. In Figure 0.2, the procedure *clnt_create* is used for this purpose. It returns a client 'handle' to use when remote procedures are called. The client handle contains the necessary information for communicating with the server port, such as the socket descriptor and socket address.

---

clnt_create → *clientHandle*

Gets a client handle. The arguments give the hostname of the server and the program and version numbers specified in the interface definition. The last argument specifies whether UDP or TCP should be used.

Several variants of *clnt_create* are available. These provide more control for the programmer such as specifying the service port. When UDP is used the time-out value between retries may be specified. When TCP is used the send and receive buffer sizes may be specified.

The remote server procedures are called in normal C language procedure call notation. The remote calls are made indirectly via calls to the client stub procedures. Remote procedures are called in the same way as local procedures with two arguments. The input arguments must be packed into a single structure which is passed as the first argument, and the ‘handle’ returned by *clnt_create* is passed as the second argument. The return value may be a structure containing several results. The client stub procedure name is the name given in the interface definition, converted to lower case and with an underscore and the version number appended, for example, READ is converted to *read_2*.

Client stubs ◊ Each client stub procedure calls the procedure *clnt_call* which has a similar function to *DoOperation* in Chapter 4.

---

clnt_call

Makes an RPC call to a server. The arguments specify: a client handle, a procedure identifier, the input arguments and a procedure to marshal them, a reference to a variable in which to store the output arguments, a procedure to unmarshal them and the total time in seconds to wait for a reply after several retries.

Clnt_call uses at-least-once call semantics. The time-out between retries has a default value which can be set when the ‘handle’ is obtained. The number of retries is the total time to wait divided by the time-out between retries. After it sends a request message, it waits for a time-out period and if there is no reply, tries again several times. Eventually if it gets no answer, it returns an error value. If an RPC is successful it returns zero. The input arguments and the procedure to marshal them are each given in a single argument to *clnt_call*. If there are several input arguments, they must be grouped together into a structure and the marshalling procedure must be designed to flatten the entire structure. If several results are returned by the server, they will be unmarshalled and stored in a single structure. When Sun RPC is used with UDP, the length of request and reply messages is restricted in length - theoretically to 64 kilobytes, but more often in practice to 8 or 9 kilobytes

Figure 0.2 C program for client in Sun RPC.

```
/* File : C.c - Simple client of the FileReadWrite service. */

#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite .h"

main(int argc, char ** argv)
{
        CLIENT *clientHandle;
        char *serverName = "coffee";
        readargs a;
        Data *data;

        clientHandle= clnt_create(serverName, FILEREADWRITE,
              VERSION, "udp");      /* creates socket and a client handle*/
        if (clientHandle==NULL){
               clnt_pcreateerror(serverName); /* unable to contact server */
              exit(1);
        }
        a.f = 10;
        a.position = 100;
        a.length = 1000;
        data = read_2(&a, clientHandle);/* call to remote read procedure */
        •••
        clnt_destroy(clientHandle);            /* closes socket */
}
```

Figure 0.3 C program for server procedures in Sun RPC.

```
/* File S.c - server procedures for the FileReadWrite service */
#include <stdio.h>
#include <rpc/rpc.h>
#include"FileReadWrite.h"

void * write_2(writeargs *a)
{
 /* do the writing to the file */
}

Data * read_2(readargs * a)
{
        static Data result;       /* must be static */
        result.buffer  =  ...     /* do the reading from the file */
        result.length  =  ...     /* amount read from the file */
        return &result;
}
```

Server program ◊ The implementor uses the C function prototypes in the header file created by *rpcgen* as a basis for the implementation of the service as shown in Figure 0.3. The server procedure names are the names given in the interface definition converted to lower case and with an underscore and the version number appended. The argument of each server procedure is a pointer to a single argument or to a structure containing all the arguments. Similarly the value returned is a pointer to a single result or to a structure that contains the results. The latter must be declared as *static*.

The server program consists of the server procedures, supported by the *main*, the despatcher and the marshalling procedures, all of which are output by *rpcgen*. The *main* procedure of a server program creates a socket for receiving client request messages and then exports the service interface by informing the local port mapper of the program number, version number and the port identifier of the server.

When a server receives an RPC request message, the despatcher checks the program and version numbers, unmarshals the arguments and then calls the server procedure corresponding to the procedure number specified in the RPC request message. When the server procedure returns the results, it marshals them and transmits the reply message to the client.

Binding ◊ We have already noted that Sun RPC operates without a network-wide binding service. Therefore clients must specify the hostname of the server when they import a service interface. The port mapper enables clients to locate the port number part of the socket address used by a particular server. This is a local binding service – it runs at a well-known port number on every host and is used to record the mapping from program number and version number to port number of the services running on that host. When a server starts up it registers its program number, version number and port number with the local port mapper. When a client starts up, it finds out the server's port by making a remote request to the port mapper at the server's host, specifying the program number and version number. This means that servers need not run at well-known ports.

When a service has multiple instances running on different computers, the instances may use different port numbers for receiving client requests. Recall that broadcast datagrams are sent to the same port number on every computer and can be received by processes via that port number. If a client needs to multicast a request to all the instances of a service that are using different port numbers, it cannot use a direct broadcast message for this purpose. The solution is that clients make multicast remote procedure calls by broadcasting them to all the port mappers, specifying the program and version number. Each port mapper forwards all such calls to the appropriate local service program, if there is one.

Marshalling ◊ Sun RPC can pass arbitrary data structures as arguments and results. They are converted to External Data Representation (XDR). The marshalling and unmarshalling procedures specified in *clnt_call* may be built-in procedures supplied in a library or user-defined procedures defined in terms of the built-in procedures. The library procedures marshal integers of all sizes, characters, strings, reals and enumerated types.

Lower-level facilities ◊ The RPC facilities described above supply a set of defaults which are adequate for most purposes. The lower-level facilities provide additional control for the programmer who needs it. The main lower-level facilities are the following:

- tools for testing the implementation and running of services, such as null RPC calls, to test whether a server is running, checks for invalid procedure identifiers and pseudo RPC calls allowing client and server to be tested within a single process;
- management of dynamic memory allocation in marshalling procedures (which is not provided by the built-in XDR procedures);
- broadcast RPC. A client program may make an RPC to all instances of a service. This call is directed to the port mapper at all computers in the local network which passes it on to the local service registered with the given program name. The client picks up any replies one by one;
- batching of client calls that require no reply. The RPC calls can be buffered and then sent in a pipeline to the server over TCP/IP;
- call-back by the server to a client. This allows the client to become a server temporarily and to pass in an RPC call the information about its service;
- authentication: Sun RPC includes a mechanism allowing clients to pass authentication parameters that can be checked by the server. The default is that this mechanism does not operate. The client program may select a UNIX or a DES style of authentication. In the UNIX style, the *uid* and *gid* of the user running the client program are passed in every request message. The authentication information is made available to the server procedures via a second argument. The server program is responsible for enforcing access control by deciding whether to execute each procedure call according to the authentication information. Chapter 16 describes DES (Data Encryption Standard). DES authentication is secure in comparison with UNIX authentication.

References

Srinivasan 1995a	Srinivasan, R. (1995). RPC: *Remote Procedure Call Protocol Specification Version 2*. Internet RFC 1831. August.