Advanced Operating System (CS G623)
First Semester 2019-2020

Objective:

1. To get familiarized with RPC (Remote Procedure Call) and working of SUN RPC
2. To learn about coding RPC using C-Programing

Data conversion:

There is a possibility of having computers of different architectures in the same network. For e.g. we may have DEC or Intel machines (which use Little-endian representation) connected with IBM or Motorola PCs (which use Big-endian representation) . Now a message from an Intel machine, sent in Little-endian order, may be interpreted by an IBM machine in Big-endian format. This will obviously give erroneous results. So we must have some strategy to convert data from one machine's native format to the other one's or, to some standard network format.

Definition of RPC:

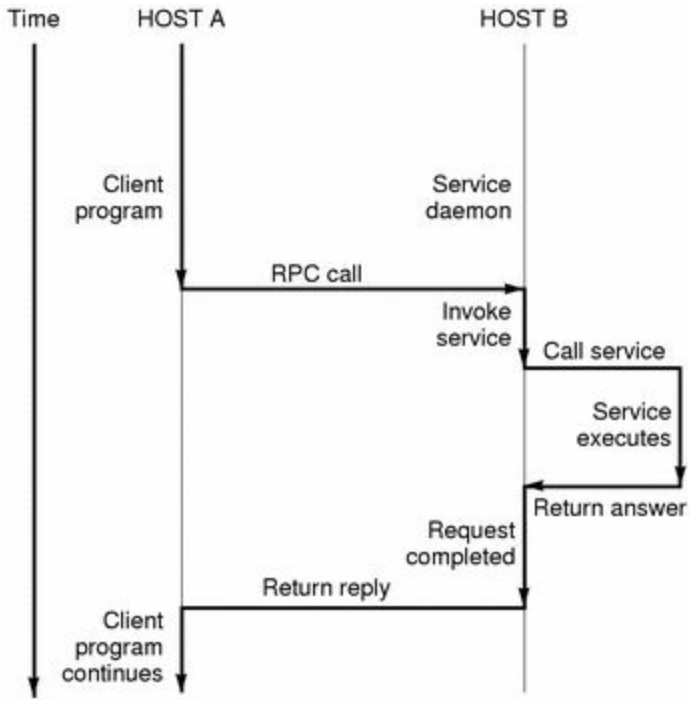
Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often does not even know they are using the network!



2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

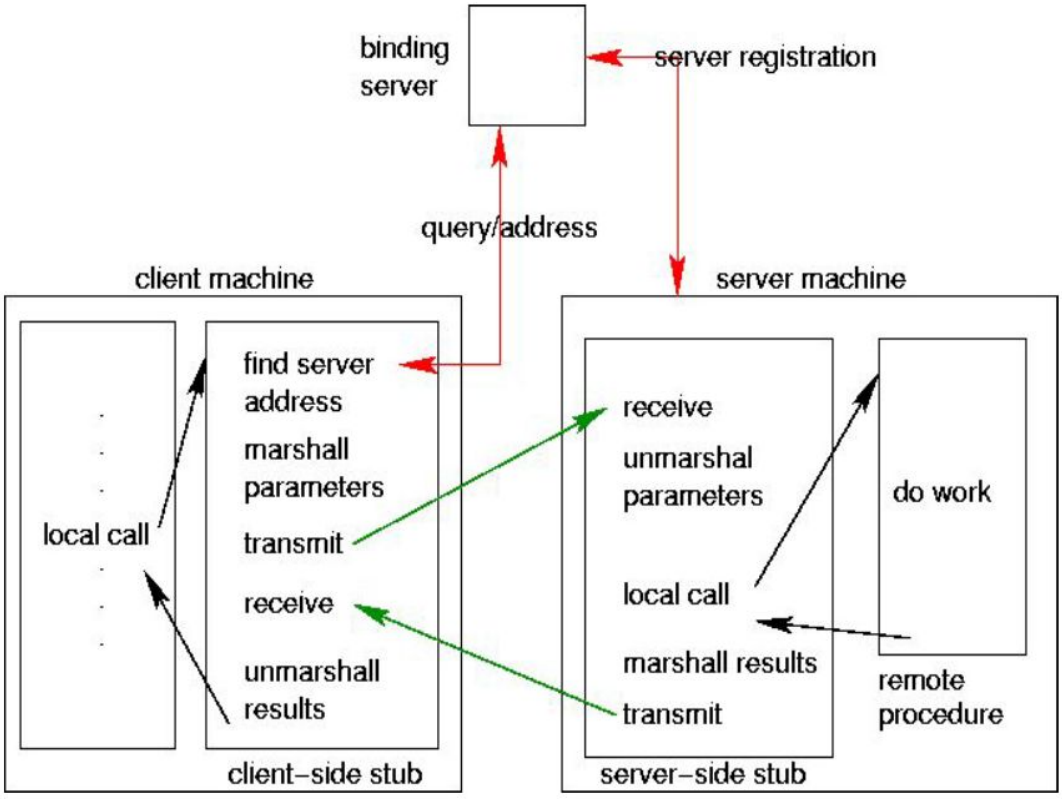
RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

The following steps take place during a RPC:

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub marshalls(pack) the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.



4. On the server, the transport layer passes the message to a server stub, which unmarshalls(unpack)the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.

6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

7. The client stub demarshalls the return parameters and execution returns to the caller.

RPC Issues :

Issues that must be addressed:

- **Marshalling:** Parameters must be *marshalled* into a standard representation. Parameters consist of simple types (e.g., integers) and compound types (e.g., C structures or Pascal records). Moreover, because each type has its own representation, the types of the various parameters must be known to the modules that actually do the conversion. For example, 4 bytes of characters would be uninterpreted, while a 4-byte integer may need to the order of its bytes reversed.

- **Semantics:**
    - *Call-by-reference* not possible: the client and server don't share an address space. That is, addresses referenced by the server correspond to data residing in the client's address space.
    - One approach is to simulate call-by-reference using *copy-restore*. In copy-restore, call-by-reference parameters are handled by sending a copy of the referenced data structure to the server, and on return replacing the client's copy with that modified by the server. However, copy-restore doesn't work in all cases. For instance, if the same argument is passed twice, two copies will be made, and references through one parameter only changes one of the copies.

- **Binding:** How does the client know who to call, and where the service resides? The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it

contacts a name server to determine the transport address at which the server resides.

    - Binding consists of two parts:
        - Naming: Remote procedures are named through interfaces. An interface uniquely identifies a particular service, describing the types and numbers of its arguments. It is similar in purpose to a type definition in programming languages.
        - Locating: Finding the transport address at which the server actually resides. Once we have the transport address of the service, we can send messages directly to the server.
        - A Server having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it. A Client must import an (exported) interface before communication can begin.

- **Transport protocol:** What transport protocol should be used

Semantics of RPC :

Unlike normal procedure calls, many things can go wrong with RPC. Normally, a client will send a request, the server will execute the request and then return a response to the client. What are appropriate semantics for server or network failures? Possibilities:

While implementing RPC, B&N determined that the semantics of RPCs could be categorized in various ways:

Exactly once: The most desirable kind of semantics, where every call is carried out exactly once, no more and no less. Unfortunately, such semantics cannot be achieved at low cost; if the client transmits a request, and the server crashes, the client has no way of knowing whether the server had received and processed the request before crashing.

At most once: When control returns to the caller, the operation will have been executed no more than once. What happens if the server crashes? If the server crashes, the client will be notified of the error, but will have no way of knowing whether or not the operation was performed.

At least once: The client just keeps retransmitting the request until it gets the desired response. On return to the caller, the operation will be performed at least one time, but possibly multiple times.

Advantages:

1. RPC provides ABSTRACTION i.e message-passing nature of network communication is hidden from the user.

2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.

3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.

4. With RPC code re-writing / re-developing effort is minimized.

5. Process-oriented and thread oriented models supported by RPC.

Types of RPC:

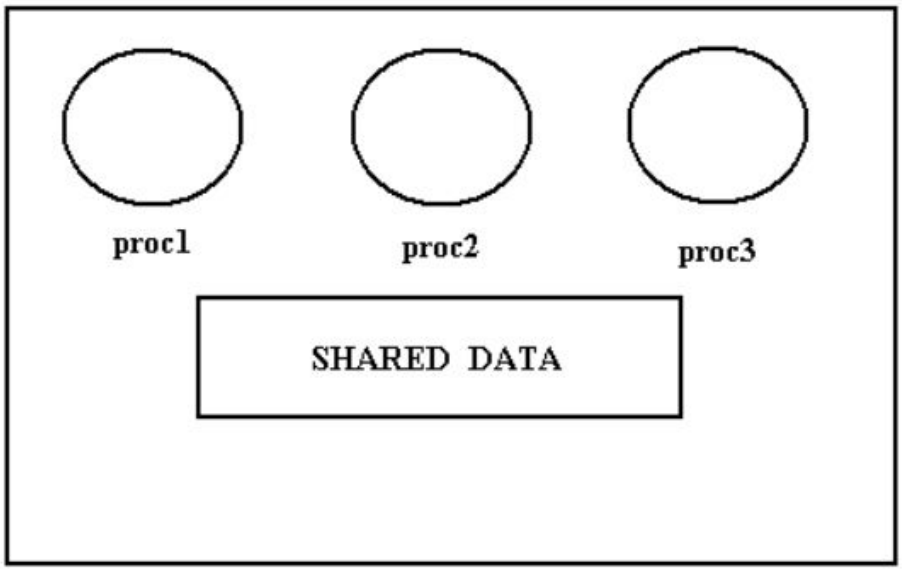
- Sun RPC
- Microsoft's DCOM
- OMG's CORBA
- Java RMI
- XML/RPC
- SOAP/.NET
- AJAX (Asynchronous Javascript and XML)

SUN RPC Model:

The basic idea behind Sun RPC was to implement NFS (Network File System). Sun RPC extends the remote procedure call model by defining a remote execution environment. It defines a *remote program* at the server side as the basic unit of software that executes on a remote machine. Each remote program consists of one or more remote procedures and global data. The global data is static data and all the procedures inside a remote program share access to its global data. The figure below illustrates the conceptual organization of three remote procedures in a single remote program.

Sun RPC allows both TCP and UDP for communication between remote procedures and programs calling them. It uses the at least once semantic i.e., the remote procedure is executed at least once. It uses copy-in method of parameter passing but does not support copy-out style. It uses XDR for data representation. It does not handle orphans(which are servers whose corresponding clients have died). Thus if a client

gives a request to a server for execution of a remote procedure and eventually dies before accepting the results, the server does not know whom to reply. It also uses a tool called *rpcgen* to generate stubs automatically.



PROGRAM

Let us suppose that a client (say client1) wants to execute procedure P1(in the figure above). Another client (say client2) wants to execute procedure P2(in the figure above). Since both P1 and P2 access common global variables they must be executed in a mutually exclusive manner. Thus in view of this Sun RPC provides mutual exclusion by default i.e. no two procedures in a program can be active at the same time. This introduces some amount of delay in the execution of procedures, but mutual exclusion is a more fundamental and important thing to provide, without it the results may go wrong.

Thus we see that anything which can be a threat to application programmers, is provided by SUN RPC.

RPC Programming :

RPC Programming can be thought in multiple levels. At one extreme, the user writing the application program uses the RPC library. He/she need not have to worry about the communication through the network. At the other end there are the low level details about network communication. To execute a remote procedure the client would have to go through a lot of overhead e.g., calling XDR for formatting of data, putting it in output buffer, connecting to port mapper and subsequently connecting to the port through which the remote procedure would communicate etc. The *RPC library* contains procedures that provide almost everything required to make a remote procedure call. The library contains procedures for marshaling and unmarshaling of the arguments and the results respectively. Different XDR routines are available to change the format of data to XDR from native, and from XDR to native format. But still a lot of overhead remains to properly call the library routines. To minimize the overhead faced by the application programmer to call a remote procedure a tool *rpcgen* is devised which generates client and server stubs. The stubs are generated automatically, thus they have loose flexibility e.g., the timeout time, the number of retransmissions are fixed. The program specification file is given as input and both the server and client stubs are automatically generated by rpcgen.

The specification file should have a .x extension attached to it. It contains the following information:-

- constant declarations ,
- global data (if any),
- information about all remote procedures ie.

- procedure argument type ,
- return type .

Sun Remote Procedure Call Mechanism:

- Originally developed by Sun, but now widely available on other platforms (including Digital Unix). Also known as Open Network Computing (ONC).
- Sun RPC package has an RPC compiler (rpcgen) that automatically generates the client and server stubs.
- RPC package uses XDR (eXternal Data Representation) to represent data sent between client and server stubs.
- Has built-in representation for basic types (int, float, char). Also provides a declarative language for specifying complex data types.

Implementing RPC :

- Procedure arguments: To reduce the complexity of the interface specification, Sun RPC includes support for a single argument to a remote procedure.

- Procedure identification: Each procedure is identified by:
  – Hostname (IP Address)
  – Program identifier (32 bit integer)
  – Procedure identifier (32 bit integer)
  –Program Version identifier for testing and migration.

- Identifiers : Each remote program has a unique ID.

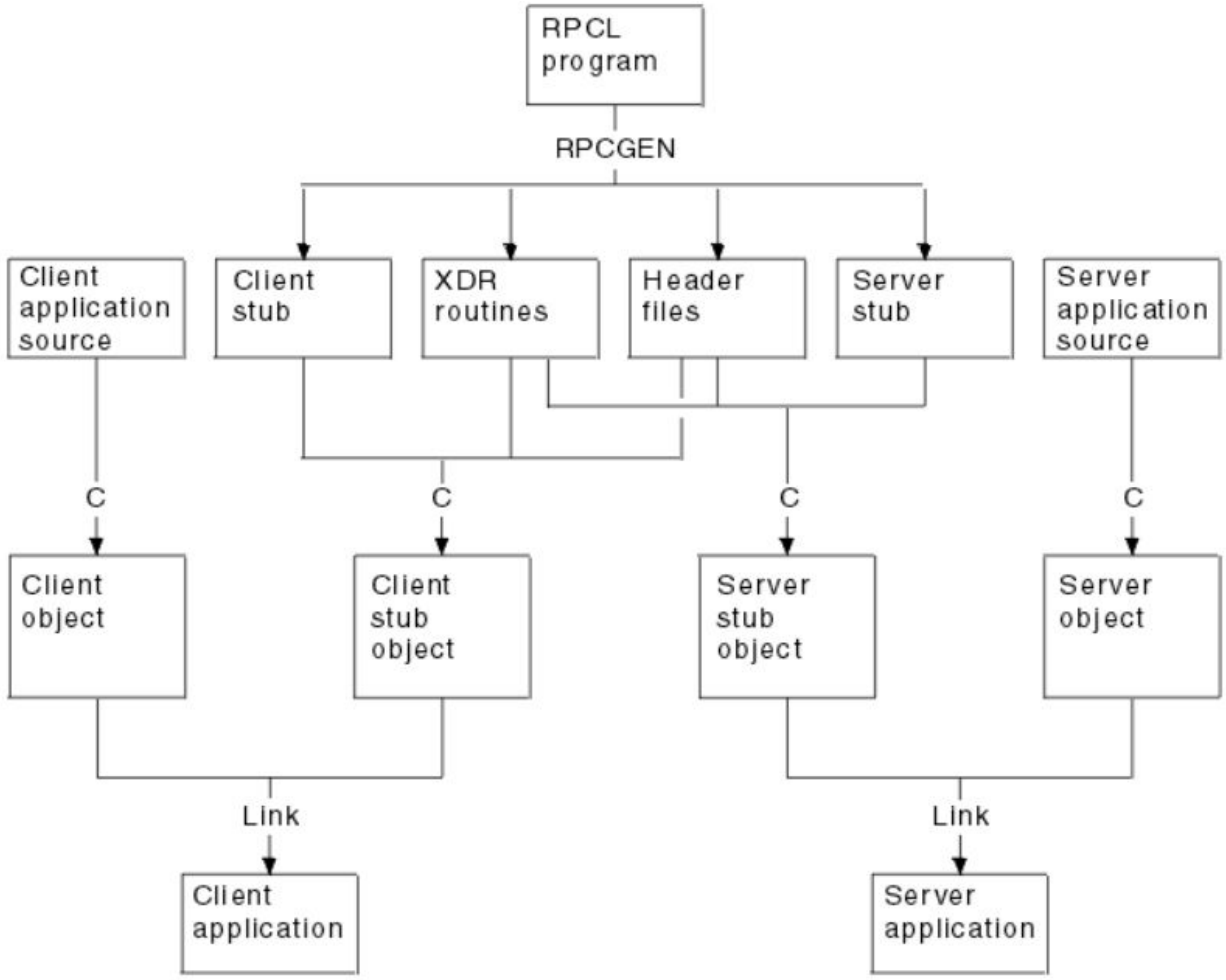
- Sun divided up the IDs:
  – 0x00000000 - 0x1fffffff Sun
  – 0x20000000 - 0x3fffffff User
  – 0x40000000 - 0x5fffffff Transient
  – 0x60000000 - 0xffffffff Reserved

SUN RPC: Interface Definition Language

```
struct square_in {
     long arg1;
};
struct square_out {
     long res1;
};
program SQUARE_PROG { version
     SQUARE_VERS {
          square_out SQUAREPROC(square_in) = 1;
          } = 1;
} = 0x13451111;
```

(IDL file: square.x)

Rpcgen Compiler:



Rpcgen continued…

```
bash$ rpcgen square.x
```

- **Produces:**
  - square.h (header server stub)
  - square_svc.c (client stub)
  - square_clnt.c (XDR conversion routines)
  - Square_xdr.c

Function names derived from IDL function names and version numbers

Square Client: Client.c

```
#include “square.h”
#include <stdio.h>
int main (int argc, char **argv)
{
      CLIENT *cl; square_in in; square_out *out;
            if (argc != 3) {
                  printf(“client <localhost> <integer>”);
            exit (1);
            }
      cl = clnt_create (argv[1], SQUARE_PROG, SQUARE_VERS,
“tcp”);
      in.arg1 = atol (argv [2]);
      if ((out = squareproc_1(&in, cl)) == NULL)
```

```
        {
            printf (“Error”); exit(1);
        }
        printf (“Result %ld\n”, out -> res1); exit(0);
    }
```

Square server: Server.c

```
    #include “square.h”
    #include <stdio.h>
    square_out *squareproc_1_svc (square_in *inp, struct
    svc_req *rqstp)
    {
        static square_out *outp;
        outp.res1 = inp → arg1 * inp → arg1;
        return (&outp);
    }
```

Exe creation:

```
gcc –o client client.c square_clnt.c square_xdr.c –lnsl
gcc –o server server.c square_svc.c square_xdr.c –lrpcsvc -lnsl
```

Another example of Remote date:

http://web.cs.wpi.edu/~rek/DCS/D04/SunRPC.html

Make Command :

1. create the square.x file
2. Then to get all the necessary files run the following command

   `rpcgen -a -C square.x`
3. Then to get the square_client.c and square_server.c run the following command:-

   `make -f Makefile.square`
4. Now change the code of the square_client.c and square_server.c according to the needs.
5. Again compile them using the command:

   `make -f Makefile.square`
6. Run the Server first Using:

   `./square_server`
7. Run the Client Using:

   `./square_client`