University of Pittsburgh

CS 2510 - COMPUTER OPERATING SYSTEMS

SIMPLE REMOTE PROCEDURE CALL

Final Project Report

Ankita Mohapatra
anm249@pitt.edu

October 17, 2016

Introduction:

Remote Procedure Call is a protocol that one program uses to request a service from another program. RPC uses client/server model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently.

When program statements that use RPC are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Results are returned the same way.

In this project, we are developing and implementing *RPC over TCP/UDP protocol*.

Main Components of SRPC package:

The SRPC package is composed of five important components. They are described as follows:-

1. Stub Generator:

A stub in distributed computing is a piece of code used for converting parameters passed during a Remote Procedure Call. Stubs can be generated either manually or automatically. In our code which is the dynamic binding code and manual, the user can create his set of stubs. In the procedure call, caller pushes parameters and returns address on the stack. There are three types of parameter passing methods namely:

- call-by-value
- call-by-reference
- call-by-copy-restore

We have implemented the first two parameter calling methods in the code.

Stubs are generated from a formal specification of a server's interface like the version, id, etc. When the server is initialized, it registers its interface by registering at a binder program with the handle like the IP address. When a remote procedure call is implemented, the client imports the interface from the binder. The advantage of dynamic binding is the location independence, the ability to balance more load, fault tolerance and authentication.

Stub generation is the action of taking the SRPC specification file and converting it into a server stub, a client stub, and appropriate header files. Creating the server and client stubs from the xml file is simply just parsing the file and converting them into relevant code for the server and client, respectively. The stub generators have functions that are essentially building blocks to creating a working server or client stub. Each building block is a portion of code that is added to the stub itself, depending on the requirements for what kind of procedure needs to be supported. The generator builds the source code part-by-part based on the configuration file.

The stub generator makes three function calls, namely the generation of the client stub, server stub and header files.

The generation of the stubs and the subsequent compilation of those stubs with client or server code follows the schematic diagram as given below:

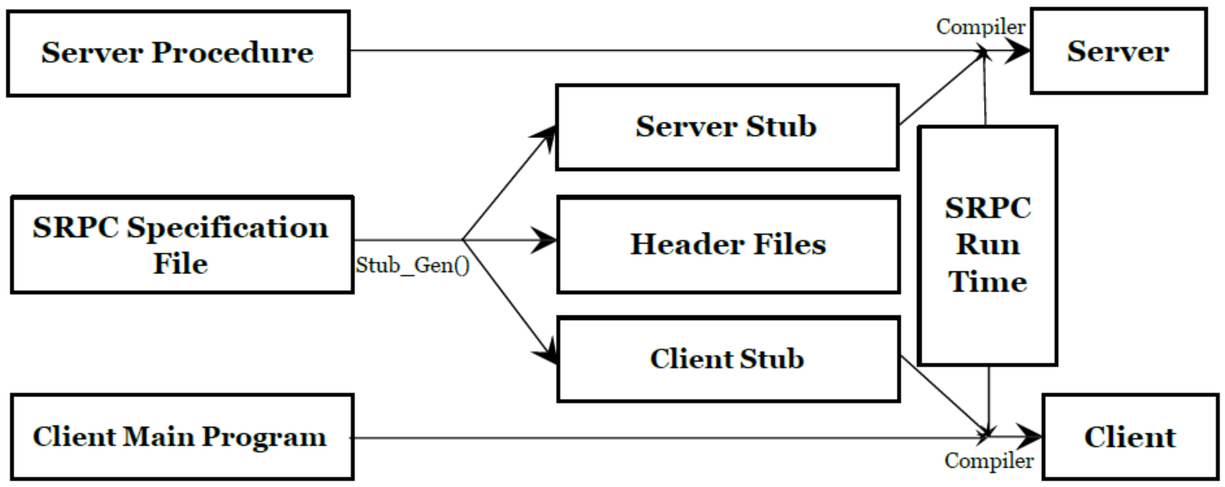

Figure 1: Generation & Compilation of Stubs

There are some assumptions that need to be made in order for the stub generator to create correct code.

Client Stub

Most of the functionality lies in the client stub. First the client will read the PortMapper/Directory Service host and IP address. Then it extracts the server host and IP address that is saved in a list. Based on the procedure, it then requests to get the server host and IP address and connects to the Server. It packs the input and sends it to the Server.

Server Stub

The server stub is the generated code from the xml configuration file that describes how the server should receive data from the client, then asks the utilities code to perform the requested function. Once the utilities code is finished, then the server stub marshals the data back to the client, depending on what they asked for.

2. Header Files:

The header files includes the definition type and the methods that are available to the client to compute and these methods are computed at the server end. These header files are included in both the client side and the server side.

3. Client:

The client is simply the machine that is requesting for a certain functionality that it is assumed to be unable to perform. The client has some data that it wishes to perform some function on, but instead uses an RPC to allow a server to do it. After contacting the PortMapper/Directory Service to retrieve the server that can handle the requested RPC, it uses the client stub to marshal the data over to that server. It then expects some data marshaling in return, which corresponds to the solution of the requested RPC.

The client is not expected to remember to which server it should request a RPC, since it is unclear to the client that the server is still in operation. So for every RPC, the client will make a connection to the PortMapper/Directory Service to determine the IP Address and the port number of the server that will handle the specified RPC.

The client code is relatively straightforward. Programming the client is simply making calls to functions, while the functions themselves as determined by the client stub does all the data marshaling. The client is hidden from these details.

4. Server:

The server is simply the host that will take requests from the client and will execute the function after the data has been sent by the client. The server without the stub is simply just a connection handler; it just merely sets up a server socket and waits for incoming connections. It then passes on the connection information with the server stub and expects the server stub to handle all data marshaling. The server also contains utility functions that allow it to perform operations on appropriate data, in the assumption that the client is unable to perform these functions itself due to resource constraints. Here we are implementing a multithreaded server.

The server upon startup should use the server stub code to register itself with the PortMapper/Directory Service. In addition, upon shutdown, the server should use the server stub code to deregister itself. For simplicity, the server is shutdown by sending it a function ID of 32768. In practice though, this is not the correct way to shut down the system.

Server and Client Preconditions and Assumptions:

There are certain preconditions that are necessary to ensure the correctness of the stub generation for the server and client:

- The server must contain all procedures that it can run.
- The server must be able to open a server socket. All RPC requests must be done on that server socket.
- The client must be able to connect to the server via sockets.
- The client and server must know the IP address and port of the PortMapper/Directory Service.
- The client and server must be able to communicate via TCP/UDP.
- The server cannot perform more than 32768 functions.
- The server and client only have one thread of execution each. This is done for simplicity.

Preconditions and Assumptions for Library Procedures

There are also certain requirements for the library procedures:

- If the function returns a complex type, the last complex parameter should be the item that will be altered to return that complex item. For example, if the function is returning an integer array, the last parameter input into the function should be that array.
- The functions should not be altering more than one parameter at a time. (This is good coding practice anyway.) The stub generators cannot generate stub code to handle more than one alteration of parameters.
- The procedures should all be assumed to be correct and optimal, but for simplicity, they are implemented with their easiest-to-code versions, which tend not to be optimal.
- Error checking for these procedures are not guaranteed. If improper parameters are provided to the functions, their behavior is undefined.

5. PortMapper/Directory Service:

The PortMapper/Directory Service is simply a publicly known location where servers can announce to the world by registering to the service that it can perform certain RPCs. In addition, clients can ask the PortMapper/Directory Service, the IP Address and port numbers for the servers that can provide the client with the requested RPC. The PortMapper/Directory Service, therefore, provides servers with registering (command = 1) and unregistering (command = 2), and provides clients with lookups on servers that can perform the specified RPC via function ID, program number, and version number (command = 3). For simplicity, it also provides a shutdown command (command = 4), but in practice, this should not be the case.

The PortMapper/Directory Service itself contains an array of linked lists, where each node represents a server that can run the given function. The function id number refers to the index into the array. The PortMapper/Directory Service then must search through the whole linked list to find the matching program and version number to send back the results to the client. This is not the best way to approach this issue, since linked lists are not optimal, but it was the conceptually easiest approach to take.

As servers register themselves with the PortMapper/Directory Service, the service will create a new node with the pertinent information and append it to the linked list. The server must create a new register connection for each function that it can support. This is simply a one-time cost during registering, and does not affect RPC performance in the assumption that servers do not continually make connections to the PortMapper/Directory Service. For unregistering a server, the server does not need to make a new connection for each function. Instead, the PortMapper/Directory Service will iterate through all nodes to remove the nodes that contain the unregistering server's information. Again, this is simply a one-time cost for unregistering and does not affect RPC performance in the assumption that servers do not continually make connections to the PortMapper/Directory Service.

The client can make a request to the PortMapper/Directory Service by providing a function ID number, a program number, and a version number. The function ID number will index into the array of server listings, which then the PortMapper/Directory Service must search through the linked list to find the first occurrence of the given program and version number. As a result, there is also no load balancing; the first server that registers a function with a given program and version number will always be the server that the client will contact.

PortMapper/Directory Service Preconditions and Assumptions

There are also certain requirements for the PortMapper/Directory Service:

- The PortMapper/Directory Service should be able to store the array of function listings in main memory.

- The PortMapper/Directory Service has only several thread of execution, meaning it can handle a sudden flood of several server registrations and unregistrations. Normally, there would be multiple threads handling this to reduce the impact these operations have on clients requesting server information, but the project does not implement this for simplicity.

Sequence of the events during an RPC call:

Assuming that the server has already registered with the PortMapper/Directory Service, the following are the sequence of events that occur during an RPC call:

The client is essentially blind to the underlying components of data marshaling. The client merely has to prepare the parameters depending on what it needs to do, then submit those parameters to the client stub as a simple function call. What comes back is simply the result of the function call, again the networking abstracted away, assuming the parameters are well-formed and expected.

Client to PortMapper/Directory Service

The following figure shows a graphical representation of a call from the client to the PortMapper/Directory Service. When the client issues a function call that requires an RPC, the client stub will first request the server information from the PortMapper/Directory Service. The client stub will send the information request command (i.e. the value 3) and the function ID, the program number, and the version number (as supplied by the XML configuration file when the stub was generated) (*Part A in figure 2*), and the PortMapper/Directory Service will respond with an IP Address and a port number to contact the server that hosts the RPC (*Part B in figure 2*).

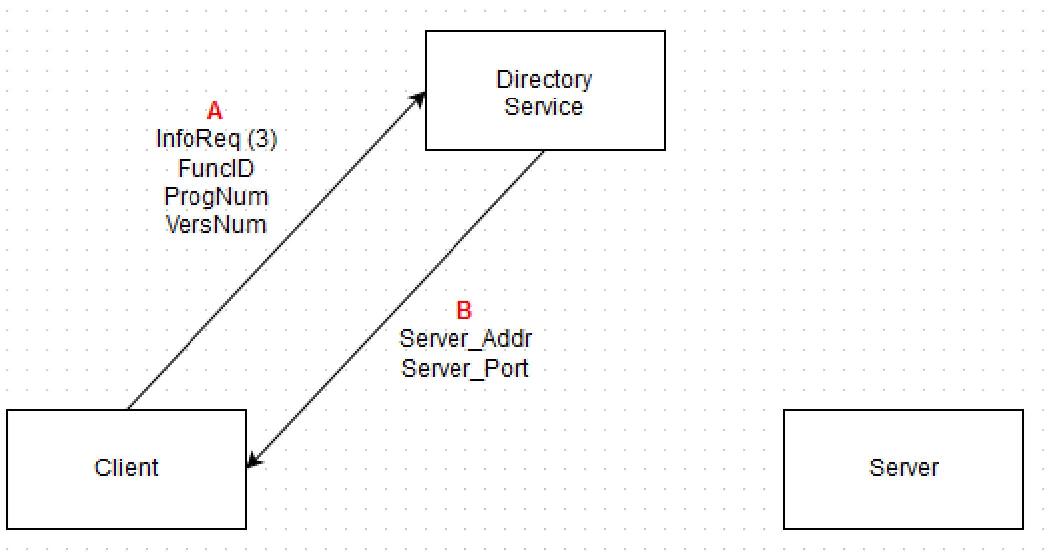

Figure 2: Client contacting the Directory Service/PortMapper

Client to Server

The following figure shows a graphical representation of a call from the client to the server. The client stub will connect to the server (as specified by the PortMapper/Directory Service) and send the same function ID number to the server. Afterwards, the client stub will begin to marshal all the necessary data over (*Part C in figure 3*). The client will wait for the server to complete the request, then receive the data from the server before returning the result (*Part D in figure 3*).

The connection is then closed. Any subsequent calls to the same RPC must undergo the same process again.

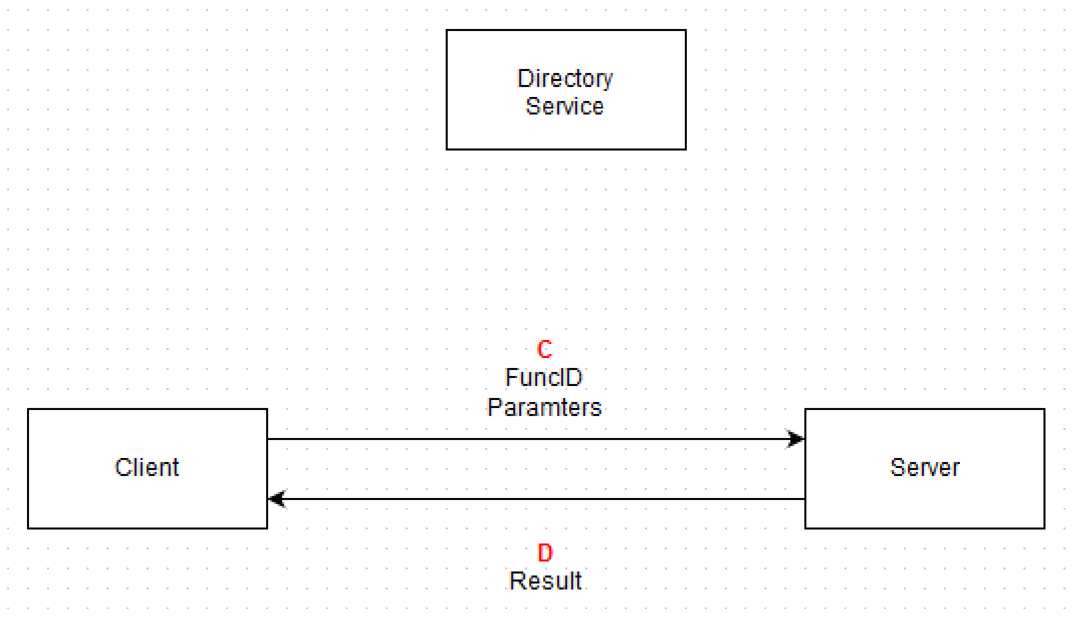

Figure 3: Client making call to RPC

Schematic Diagram showing sequence of the events during an RPC call:

The flow diagram for the Binding and Execution is as below:

I) SRPC Binding and Execution:

The server first registers its details to the PortMapper/Directory Service. The Directory Service/PortMapper stores the program number and program version. After registering itself with the Directory Service/PortMapper, the server moves to wait for the client requests. The client stub generates an RPC request message containing the procedure's ID and marshals the arguments into the request as well. The client stub then sends the request to the server at the IP address provided by the directory service.

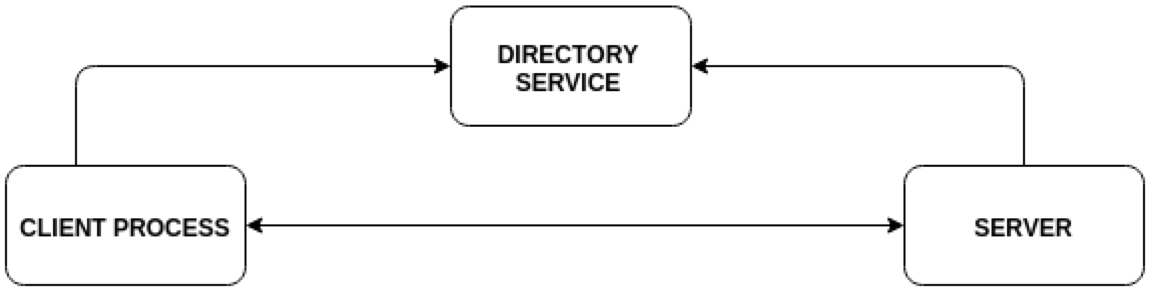

Figure 4: SRPC Binding and Execution

II) SRPC Execution Environment:

The SRPC Compiler is responsible for reading the Specification file which is an xml. This Compiler generates the header files for the client and server, client stub and server stub. The client and server include the header files. The general input details are maintained in the client and the authorization of running the specific procedure is done in server. The client stub includes the marshaling of the data and sending the details to the server. The server stub accepts the details from the client, unpacks them, computes them and then sends the details back to the client. The details are sent over the network in a marshalled form. Marshalling of the result is carried out by the server stub. The Network object has all the common procedures to connect to a particular host and port. So first the client and server connect to the Directory Service/PortMapper through the connect function. After compiling all the files, the particular object is created. So only by linking the client, client stub and network, a unique client object is created. Also by linking the server, server stub and network, a unique server object is created. Both of them exchange messages by sending and receiving the message.

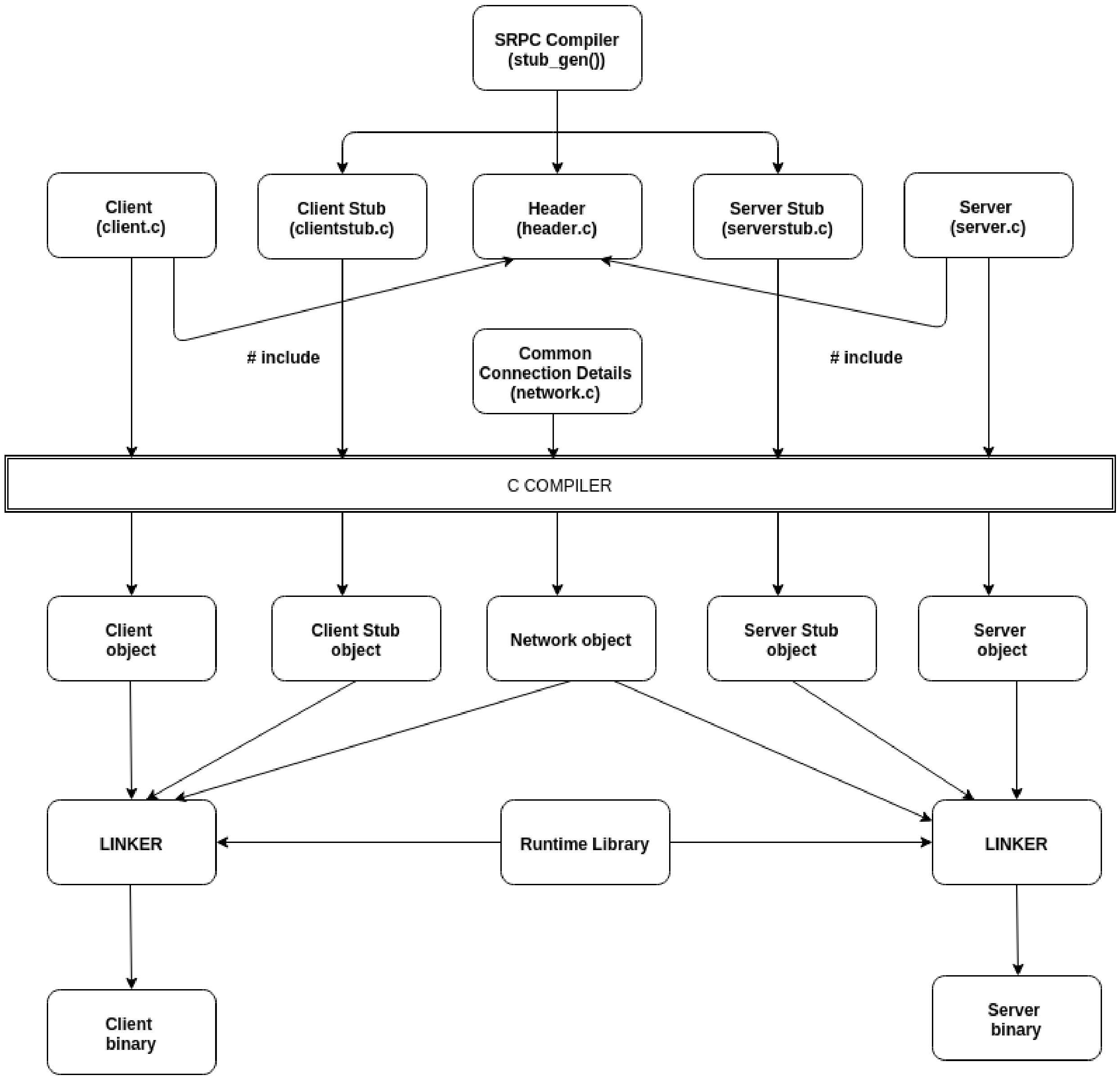

Figure 5: SRPC Execution Environment

Procedures used in implementation of SRPC package:

The server supports a simple scientific library comprising:

- **Matrix multiplication**: **Multiply**(A[n,m], B[m,l], C[n,l]),
- **Vector sorting**: **Sort**(v_i , $0 \leq i \leq n-1$),
- **Compute minimum**: **Min**(x_i , $0 \leq i \leq n-1$), and
- **Compute maximum**: **Max**(x_i , $0 \leq i \leq n-1$).

Each of the procedure has several parameters attached with it. The **tags associated with the procedure** are as follows:

- *Name* (the procedure name)
- *Id* (procedure id)
- *Type* (the type returned by the procedure)

The **tags associated with parameter** are as follows:

- *Name* (parameter name)
- *Index* (the order in which they are called)
- *Id* (the id of the parameter)
- *Type* (the datatype associated with the parameter)

1. <u>Directory Service/PortMapper</u> (portmapper.c):

First the directory Service get its IP address and port details. Then it writes its details in a file "portmapper_ip" so that it can be read from both the client and the server. It then opens its TCP/UDP socket listener to registers the servers and attend the client. The procedures used for Directory Service are:

a. write_portmapper_ip:
This function writes the IP address and the port details of the Directory Service, so that both the client and server can read the details.

b. run_directoryservice:
It first listens to the socket, declaring willingness to accept connections. It then accepts requests from server and client. A specific token is specified to recognize if the request is from the server or the client. This token is marshalled at the server and client end along with the other input data and then sent to the directory Service. If the request is recognized as the server then it allows the server to register its address and port details to the Directory Service. The details are saved in a linked list. When the request is recognized as client, then the server details are sent to the client so that it can connect to the server.

2. <u>Server</u> (server.c):

The server carries out three main steps: Firstly, it gets the Directory Service address and TCP/UDP port details. Secondly, it makes a call to compute the functionality that is requested by the client. Thirdly, it asks the server to run. The procedures used for Server are:

a. sendRegister:

This function lets the server register the service over the Directory Service and recognizes the function of each call and sets a service number. The main functionality is present in the "*serverstub.c*".

b. ***runServer:***
This function runs the server,i.e., it opens the listening socket to accept the client request. It also unpacks the input data. Based on the input, it then computes that particular functionality by executing the corresponding procedure.

c. ***doMethods :***
There are separate procedures for each functionality, thus a total of 4.

3. <u>Server Stub</u> (serverstub.c):

The procedures used for Server Stub are:

a. ***readPortMapperIP:***
This method get the details from the *portmapper_ip.txt* file.
b. ***runServer:***
This method is responsible for opening the listening socket to accept the client request. The client request is accepted and the message is saved in a character buffer.
c. ***sendRegister:***
This method connects with the Directory Service and registers the address, port and service details with the Directory Service.

4. <u>Client Stub</u> (clientstub.c):

The client stub is responsible to connect to the Directory Service, get the server details, connect to the server, marshall the data and send the data to server. The marshalling of the data is done by combining all the input as a string and passing it across the network. The procedures for Client Stub are:

d. ***readPortMapperIP:***
This method get the details from the *portmapper_ip.txt* file.

e. ***getServerAddress:***
This method gets the server details from the directory service. First it gets the address and port details that is read from the directory service. It then connects to the Directory Service through the method "*connectTCP_server*" whose functionality is provided by the "*network_function.c*". It then gets the address and port details of the server, so that it can connect to the Server.

f. ***doRPC_client:***

This method connects to the server through the method "*connectTCP_server*" whose functionality is provided by the "*network_function.c*".

g. **methods: min, max, vector_sort, matrix_multiply:**
These methods perform the basic functionality of getting the server details from the Directory Service. If the service has been registered with the Directory server only then it allows the client to connect with the server. Else it throws an error message.

5. **Client (client.c):**

The client is responsible to get the input from the user. The operation is requested by the client and the particular input is accepted and sent to the client stub to carry out other functionality.

6. **Network (network_function.c):**
This class has all the common methods that is needed for creating a socket and connecting to the server. The methods present in this file are:

a. ***createTCP_server:***
This method is used to create a TCP socket and get its address and port number. We also select TCP to be the default transport protocol in this method. This function returns a small integer descriptor which can be used to identify the socket in all future function calls, especially in the calls to connect with the server and read from the server. To find out what port number has been assigned to the socket, we call getsockname().The getsockname() function has the following syntax:

int getsockname(int socket, struct sockaddr *address, socklen_t *address_len)

When called, the function retrieves the locally-bound name of the specified socket, stores this address in the *sockaddr* structure pointed to by the address argument, and stores the length of this address in the object pointed to by the *address_len* argument. Post this it can start accepting requests from clients.

b. ***connectTCP_server:***
This function is used to connect to the remote server.

c. ***createUDP_server:***
This method is used to create a UDP socket and get its address and port number. This function returns a small integer descriptor which can be used to identify the socket in all future function calls, especially in the calls to connect with the server and read from the server. To find out what port number has been assigned to the socket, we call getsockname().The getsockname() function has the following syntax:

int getsockname(int socket, struct sockaddr *address, socklen_t *address_len)

When called, the function retrieves the locally-bound name of the specified socket, stores this address in the *sockaddr* structure pointed to by the address argument, and stores the length of this address in the object pointed to by the *address_len* argument. Post this it can start accepting requests from clients.

d. ***connectUDP_server:***
This function is used to connect to the remote server.

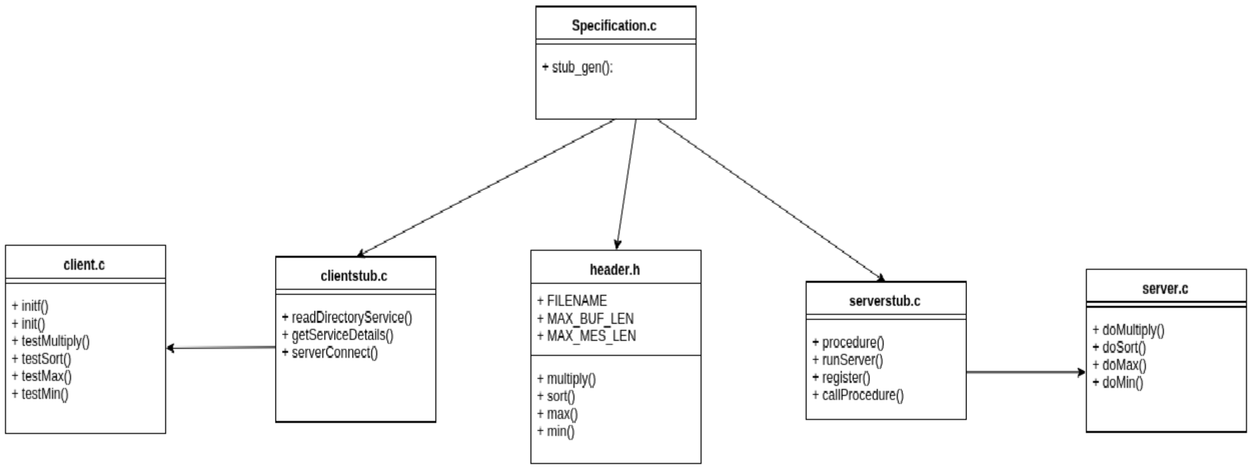

Figure 6: Class Diagram

Mechanism of Interaction:

The interactions between all of the components are essentially "stateless," where the connection is kept alive only in the duration of a single command. Whether it is registering the server to the Directory Service, or the client issuing an RPC to a server, once the connection is terminated, the service is finished. More details about how to interface each portion of the RPC service is provided in the following subsections. All connections are done over TCP using sockets.

1. Directory Service/PortMapper

When a client or server connects to the Directory Service/PortMapper, the Directory Service/PortMapper waits to receive a command value in the form of an integer. The following list indicates which value maps to which command:

1. Register Server
2. Unregister Server

3. Request Server Information
4. Shut Down

If a server sends the command value of 1, then the Directory Service/PortMapper will also wait for the server to send the function ID number, program number, the version number, and the port number that is used to listen for connections, in that order. The Directory Service/PortMapper will update its internal structures by adding a node at the appropriate function ID number index location at the end of the list, then return 1 to the server to indicate a success. Then the Directory Service/PortMapper will terminate the connection.

If a server sends the command value of 2, then the Directory Service/PortMapper will also wait for the server to send the program number, the version number, and the port number that is used to listen for connections, in that order. The Directory Service/PortMapper will update its internal structures by removing all nodes that correspond to the server with the program number, version number, and port number, then return 1 to the server to indicate a success. Then the Directory Service/PortMapper will terminate the connection.

If a client sends the command value of 3, then the Directory Service/PortMapper will also wait for the client to send the function ID number, program number, and the version number. The Directory Service/PortMapper will then index into the internal data structure depending on the function ID number, then search the nodes for those that match the program number and version number. If such a node is found, the address and port number held in the data structure corresponding to the server that can handle the function is sent back. Otherwise, the value -1 is sent back. Then the Directory Service/PortMapper will terminate the connection.

2. Server Stub

When a client connects to the server, the server waits to receive a function ID number in the form of an integer. If the server does not support such a function, meaning the function ID is not found, then the server will immediately return -1 and terminate the connection. Otherwise, the server will call the appropriate function to marshal the correct data in and call the corresponding utility function before marshaling the answer back. Then the server will terminate the connection.

If the server receives a function value of 32768, the server will return the integer 1 to the connector, terminate the connection, and then terminate itself. In practice, this is not the correct way to shut down the system.

3. Client Stub

The client merely initiates connections, but does not receive any. See the preceding sections for descriptions of how the client interacts with the Directory Service/PortMapper or the server.

Message Format:

The server and client (stubs) communicate with each other by connecting over TCP/UDP. The data that is sent is first converted into characters before being sent over the network. While understandably this is not the best way to do this for non-character based items, it was the easiest way for debugging purposes. Requests are simply just making the connection. Once the connection is made, the receiving party waits for a command value, which then determines what additional information is required. No handshaking or other security or reliability protocol is implemented for simplicity.

Request /Reply Message exchanged among the remote Hosts:

The following schematic diagram shows the request/reply messages exchanged among the Client, Directory Service/PortMapper and Server:

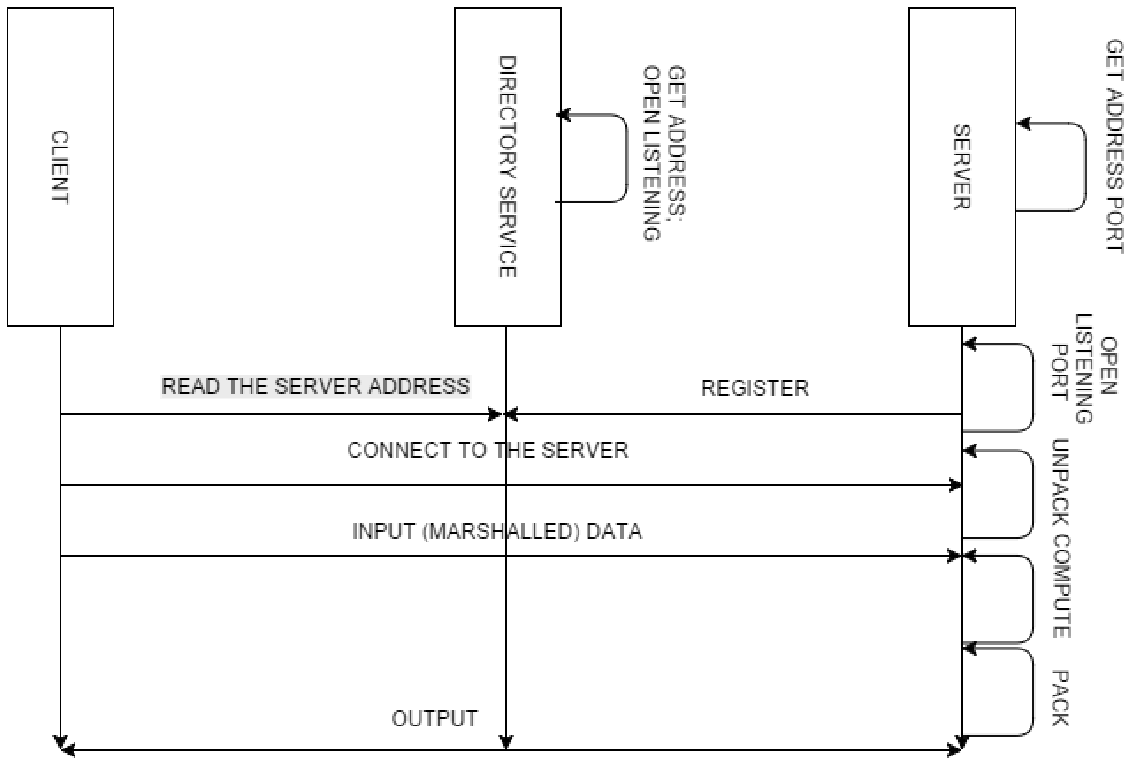

Figure 7