Project 1 Report

The model that we intend to implement for your SRPC system has five major components which are namely the client file, the client stub, the server file, the server stub and the directory service.

High level working of the model:

We first run the directory service on a known ip address and port number. Next when the server is run it registers itself with the directory service with details such as the service it is offering and the version of the service it has. It also sends data such as its ip address and the port it is receiving the connection to the client on. Once this data is present in the directory service the server can also choose to deregister itself at which point the server’s information would be removed from the directory service. The client, when it is first executed, looks to the directory service to find the connection details of the server that provides the functions that it is interested in. Once the directory service returns this data to the client it establishes a direct connection with the server using TCP. The client then sends the (marshaled) data to the server (client stub communicates with server stub) which then sends it to the server after correctly reassembling it. The server then processes the data and the output is sent to the server stub which again marshals it and sends it to the client stub. The client stub then does the role of unmarshaling the data and passing it to the client code which display it to the user.

Client:

The client is a .c file that we have included in the main which resides in the client stub. The implementation of such a system was a challenge to us as we are used to thinking of the client file as containing the main and the client stub being included into the client file. We thus have the client stub calling a function in the client file which asks the user which function they would like to execute. On getting a valid input from the user the client sends this data the client stub. The client code has the menu hardcoded into it which gives the user to select which function to execute. The reason the menu is hardcoded is because the client code is usually independent of the RPC since RPC is only responsible for sending and receiving the right data while maintaining a level of transparency.

Client Stub:

The client stub contains all the code that would be used to marshal and unmarshal the data between the client and the server. There are many functions implemented in the client server which enable this form of communication at ease and in a way which can be easily automated. The client stub, when looking for the connection information of the server, sends the directory service the version number and the program id it is looking for. This approach has been chosen as it limits the amount of information that has to be exchanged between the various components while at the same time maintaining the functionality required. The client stub also communicates with the server stub using msg-ack format. That is for every message received it sends back an ack to the server. There are two reasons for choosing this approach. On a scalability sense it ensures that the messages sent between the client and server do reach their destination and that each component has a sense of which messages have been received by the other component. The second (more practical) reason for choosing this implementation is because the recv command glitches when multiples recvs are called simultaneously. Thus waiting for an ack servers as a buffer time for the other component which ensures that its output message queue would have been flushed by then.

Directory Service:

The directory service is implemented with thread functionality. Our implementation assigns a thread of each incoming connection. We then ask the connecting component whether it is a client or server.

If it is a server we then ask it whether it wants to register or deregister. If it chooses to register then we ask it the port number, program id and version number. We then store this data in a linked list with the ip that we extract from the connection. We chose to extract the ip from the connection itself due to two reasons. Firstly, on a scalability sense this allows lesser messages to be passed between the server and directory service which in turn reduces the load placed on the communication channel. Secondly, the more practical reason we chose this design decision is since we can be confident that the ip the server is connecting to the directory service would also be the ip it uses to connect with the client. It was also noticed that it is much easier to extract the ip address from the communication link set up rather than send the data to the directory service which would involve additional marshaling and processing.

Now we reach the scenario where the server wants to deregister. In that case we only accept the port number from the server and kill all services that are assigned to that ip and port number. Again, as stated above, the ip is extracted from the communication link for the reasons stated above. The reason we chose such a design decision is because each service will be linked to a unique port number and thus we can be sure that when a server wants to deregister a particular service the only relevant information we would need is the port number. Once we register/deregister a service successfully, the server is sent a message which then confirms to it that the action was completed. After either registering or deregistering we close the connection between the directory service and the server. This is because we want a minimal number of connections open at the directory service and this is important on a scalability perspective where multiple servers would be connecting to the same directory service.

We now consider the scenario where the component connecting with the directory service is a client. In this case we ask the client the program ID and version. This information is then used to traverse the linked list to find the service related to this request. Once such a service is found, the ip address and port number of this service is returned to the client which then uses this information to set its connection with the server.

It must also be noted that when a service deregisters itself we suspend all clients associated with that service after the execution of the current action it is processing. This is done in order to ensure that we maintain consistency and reduce the recovery actions that would normally be associated with a server shutting off midway.

Server stub:

The server stub contains the various functions associated with the marshaling and unmarshaling of data on the server side. Its implementation is close to that of the client stub. One point of information to note is that the server stub acts as a multi-threaded server accepting connections when facing the client and acts as a client when facing the directory service. The server stub also has a separate thread that is specifically associated with the deregister service. Once the input is passed to this thread to deregister it automatically deregisters the service by sending the appropriate messages to the directory service and signals the other threads (by use of a global variable) to stop accepting messages from the client after completing the current action it is processing. The logic for this design has already been explained in the directory service aspect.

The biggest challenge we faced while implementing the server stub was using functions to accept the various input parameters. We realized that once the functions terminated even if they returned pointers (to arrays or other objects) they would be useless to the calling function as the scope the variable was declared in would be terminated. To get around this we sent the function pointers to data types we created in the calling function.

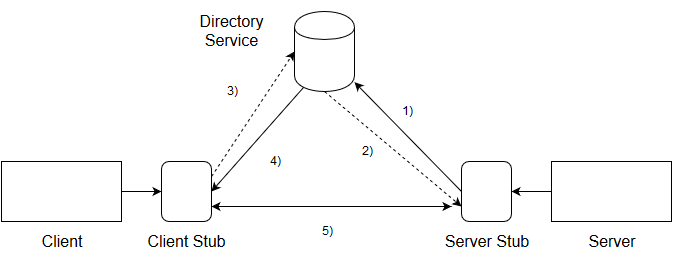
Another significant challenge was modularizing each aspect in such a way that a compiler could easily regenerate the code necessary for marshaling. This was solved again by using pointers and multiples layers of functions which served as abstraction layers and thus enabled the compiler to create the necessary code by simply placing a few related functions and the variables associated with them.

Server:

The server just contains the functions necessary for implementation. The server stub passes the unmarshaled parameters to the respective function call in the server which then uses this data to process the necessary information. The result of this operation is then passed back to the server stub which then marshals and sends this data to the client (via client stub). Again the biggest challenge here was with the idea that the main would exist in the server stub and not in the server. To implement this design we had to make regular calls from the server stub to the server with the required parameters. Yet this aspect was much simpler than the implementation of the client as the client is also responsible for taking input from the user while the server simply has to process this information.

Diagrams describing the various scenarios:

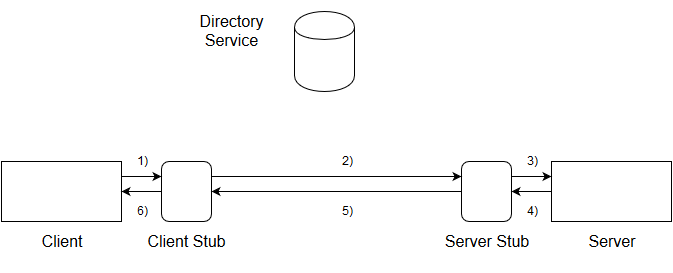
Scenario 1:



This diagram shows the initialization and the basic routes of communication. For this example we do not consider the communication between the client and server as a major point of communication and are thus not numbered.

1. Server stub sends a Registration message to the Directory Service
2. Directory Service responds by confirming that the server has registered the service
3. Client stub asks the Directory Service to provide it with connection details of a server providing a particular service
4. Directory Service responds by sending the IP and Port of the respective Server. If such a server does not exist then it sends a message informing the same
5. Assuming the client got the server details it then sets up a direct communication link with the server via the server stub. This communication link is then used to send messages between the client and server

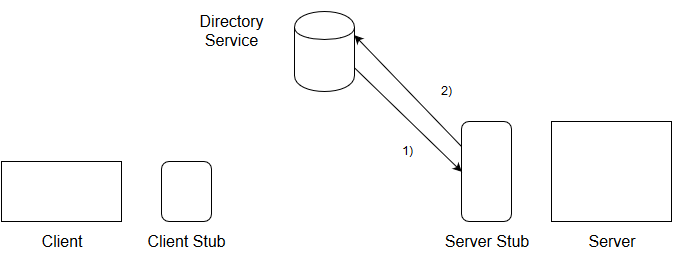
Scenario 2:



In this diagram we see the normal flow of communication between the client and server

1. The client sends the remote procedure call which calls the client stub
2. The client stubs marshals the required parameters and sends it to the server stub
3. The server stub accepts this data, unmarshals it and passes it to the server
4. The server processes the information based on the function number chosen and returns a value to the server stub
5. The server stub then marshals this data and sends it to the client stub
6. The client stub unmarshals the data and sends it to the client

Scenario 3:



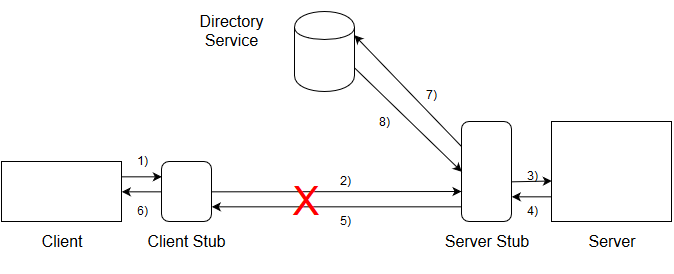
We now see the scenario where the Server is registering its service with the Directory Service via the server stub

1. The server stub sends a registration message to the Directory Service. With this message it also includes data such as the port number that the service is listening on, the program ID and version number of the service.
2. The Directory Service sends an acknowledgement or rejection based on the status of the deregistration attempt. If the Directory Service sends an acknowledgement acknowledging that it received the request then it has registered the server.

Conversely the same diagram can be used to show the scenario for server stub deregistering its service

1. A server stub sends a message to the Directory Service indicating that it wants to deregister its service. The additional information it would include it the port number associated with this service.
2. The Directory Service sends an acknowledgement or rejection based on the status of the deregistration attempt.

Scenario 4:



In this diagram we tackle the scenario where the Server stub decides to deregister itself during active communication between client and server.

The message from 1-6 are same as shown before and are thus not mentioned below

1. The server stub sends a deregistration message to the Directory Service along with the port number
2. The Directory Service responds with an acknowledgement or rejection of its attempt based on if it succeeded.

* If the Deregistration attempt succeeded then the Server Stub will close connection with the client stub after the processing of the current active procedure call