

CprE 5500 RPC Project Design Document

This project set out to create two distinct services that utilize RPC to communicate with one another. It consists of a client application that is able to call remote procedures on the server application. With the nature of RPC the client application and the server application can be located on different hosts and still communicate with one another. In this project, the client has the ability to request four different types of data from the remote server. The client can request time information, CPU loading, memory usage, and the number of active processes on the remote server. The server will collect this information in the background and will return it to the client whenever it requests it. This design document will take a closer look at how both the client service and the server service are architected along with the RPC middleware that allows them to communicate with one another.

Client

The client application consists of three distinct steps. The first step is collecting input from the user about what type of information they would like to collect from the remote host. I have decided on a multi-level approach that has the user first select what type of data they would like to receive than it asks the user again to select the timeframe that they care about for the selected data. Here is an example of the top-level interface that the user sees along with the second level interface that they would see after selecting memory usage.

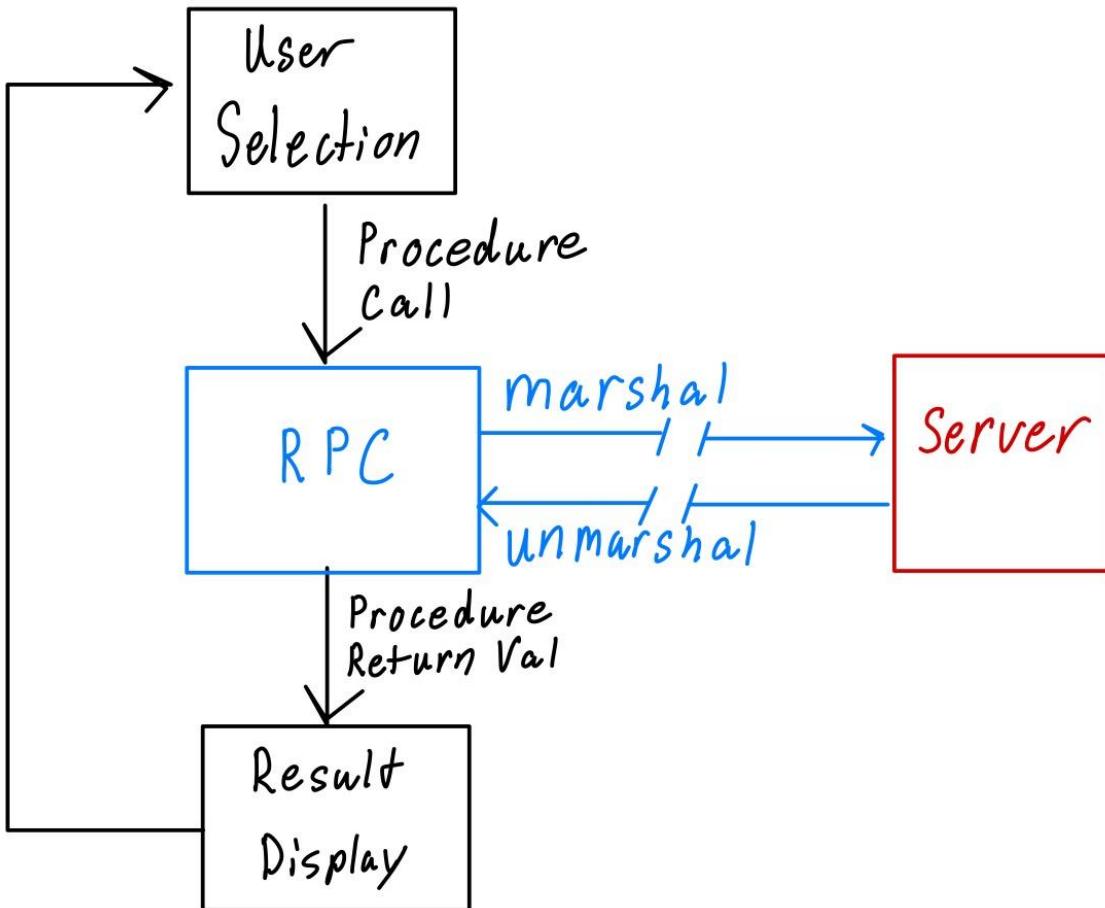
```
=====  
Menu:  
-----  
1. Time  
2. CPU Usage  
3. Memory Usage  
4. Average Processes  
5. Quit  
-----  
Choice (1-5):3  
=====  
=====  
Memory Usage Menu:  
-----  
1. Current  
2. Last Minute  
3. Last Hour  
4. Last Day  
-----  
Choice (1-4):1
```

The next step for the client would be to take the response from the user and parse out which remote procedure that it needs to call on the remote machine. Once it parses out which procedure to call, it can load the needed parameters for the procedure and use the interface provided by the middleware to call it. From there, the client will block and wait for the middleware to return the response from the remote server. (The middleware section will go into more detail about what happens once the procedure is called).

Finally once the middleware returns a value the client can display it to the user and then start the cycle over again with a new prompt to the user.

```
=====
Memory Usage Menu:
-----
1. Current
2. Last Minute
3. Last Hour
4. Last Day
-----
Choice (1-4):2
=====
8449641 KB used
8450    MB used
8.450   GB used
=====
Menu:
-----
1. Time
2. CPU Usage
3. Memory Usage
4. Average Processes
5. Quit
-----
Choice (1-5):
```

Here is a diagram of how the client service is configured.



Server

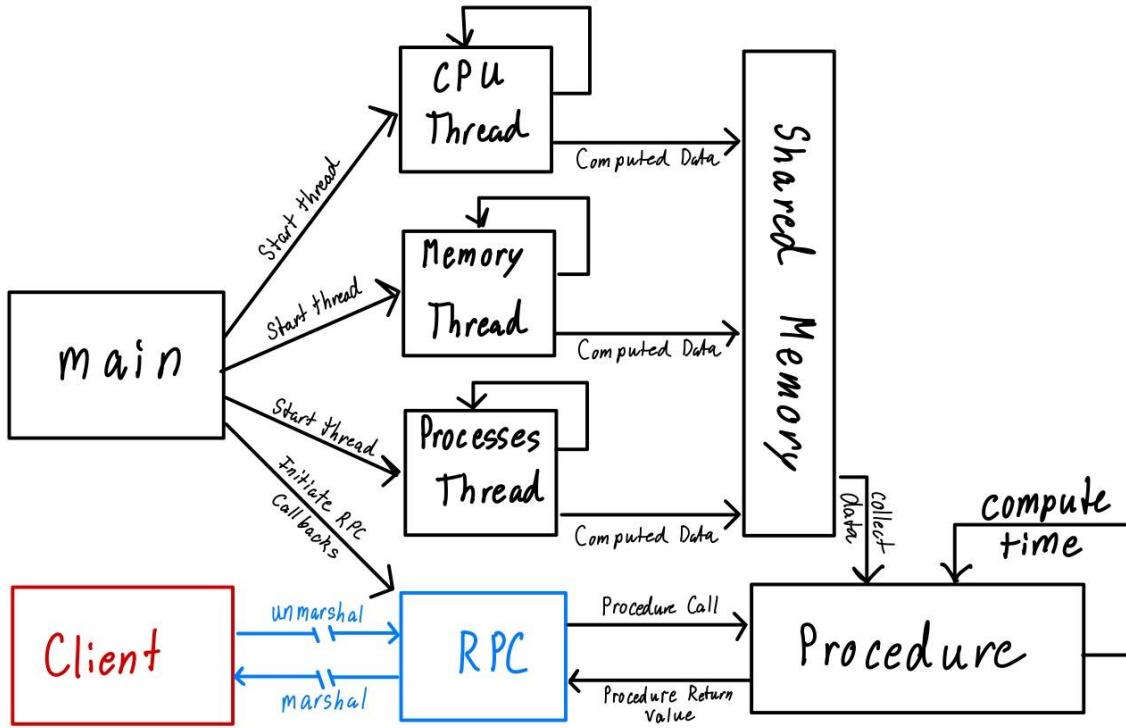
The server service that I implemented here is responsible for two things. The first of which is to compute the statistics and state of the host while it is running and to keep a history of how it is performing. The second responsibility is to field the client's remote procedure calls and return the proper information to them. These two are connected, which allows the client to receive historical information about the remote host.

Typically, when using RPC the server service is only processing when a client connects to it and requests a specific procedure to be run. I however wanted to be able to track loading and usage metrics over time with the server service which required me to take a slightly different approach. Normally the “main” for the server service is provided by the middleware and is just used as is. However, in this project I modified it to be able to

launch some separate computation threads that would track CPU and memory usage along with the number of active processes in the background. These threads would allow me to collect information and run computations in the background even when the client was not calling in to the remote server service. I acknowledge that this is an atypical approach, but it was the best way to keep a history data base of these metrics without running a second service on the host machine. The three threads that are created are for CPU loading, memory usage, and number of active processes. The CPU loading thread uses the /proc/stat file to calculate the amount of time the CPU is active vs idle. The memory usage thread uses the /proc/meminfo file to calculate the amount of free vs used memory. Finally, the active processes thread uses the /proc directory to count the number of active PIDs in the system. These computation threads hold a shared memory location that the second part of this service can also access.

The second part of the server service is the section that manages the incoming procedure requests from the client. After the middleware hands off the procedure with the desired parameters from the client, the server service does one of two things. It can make a direct computation such as in the case of the time metrics where it can direct the time or the uptime from the /proc/uptime file. It can also make a call to the shared memory to access the values computed by the separate computation threads and use the values stored there. Using either method, the final value is returned to middle ware to be sent back to the client.

Here is a diagram of how the server service is configured.



RPC Middleware

The RPC middleware is responsible for taking the normal procedure invocation from the client and marshalling it before sending it over the wire to the RPC middleware running on the remote machine. Once there, the data is unmarshalled and given to the server as what appears to be a normal procedure invocation. It then takes the returned value from the server and marshals it before sending it back across the wire to the original machine. The local middleware then unmarshals it one last time and hands the value back to the client like a normal return value. The RPC middleware in the project consists of three main files that are generated from the .X RPC definition file. The first is the .h header file that is used by both the client and the server as the definition for the remote procedures that each side is responsible for handling. The second file is the client specific _clnt.c file that has the client-side implementation for the common method found in the header file. It is responsible for marshalling the procedure and unmarshalling the returned values. The last

file created is the `_svc.c` file which contains the server-side implementation for the common methods found in the header file. It is responsible for unmarshalling the incoming procedures and marshalling the returned values from the procedures.

Conclusion

Together these three sections complete the first RPC project.