System Manual

Salient features

Marshalling of data:

The argType array is compact and packed with information. But storing data in a clever form makes accessing it and doing intuitive computations hard. As such, I broke down each function and its args into lightweight objects. This is the object diagram:

```
Func -> name
     -> vector of Args
Arg -> input
    -> output
    -> type
    -> arraySize/scalar
```

This made it very easy to do quick comparison based equality testing of functions by comparing function name, argument types, # of arguments etc. I stored these in the databases maintained by both the binder and the server. The argTypes array provided information regarding marshalling and unmarshalling the actual args parameter which was passed in as void**. The information encoded in the argTypes array would be used to calculate the size of the args and to ensure that we allocate enough space in the heap for the various data types being passed around.

Binder database, function overloading and round robin scheduling

The binder database's main purpose was to provide a mapping between the functions being registered with it and the server/port combinations that were serving these methods. For this, I maintained a vector of functions registered and another vector of servers that serve the exact same function. This ensured that I could accommodate for function overloading in the situation where more than one server is serving the exact same function with the same signature. Every time a new function is registered with the binder database, the binder will add the function to the vector of functions (Func data type) that it is already tracking. At the same time, it also maintains a vector of circular lists of server/port that are serving that very function. The indices of the functions registered in the vector maintained by the binder database match that with the vector of servers. I have an object called ServerPortCombo that holds the server and port number of a registered server in an easy to consume form. Furthermore, there's a circular list of servers that are serving a function. I chose circular lists to store the servers as a ways to enforce a round-robin style server selection process for serving requests to functions.

I implemented circular lists by using the std::list that C++ STL provides and then moving a server to the back of the list every time it was done serving. Unless it is the only server capable of serving a particular function, this will ensure that every server registered to serve gets a chance.

The binder is always listening to connections from the client or the server in a multi threaded manner so that blocking reads and writes don't hog all available resources. When it receives a termination message, it sets a termination flag within itself to indicate to all the threads that the binder is going to shutdown soon. It then sends the termination message to all the servers registered with the binder and then starts self destructing. Note that I wasn't able to complete the server side implementation of termination since I ran out of time; though I have a method that verifies if an incoming message on a socket is coming from a particular peer/port

The error codes that I was able to implement formally are in the response_codes.h file

During server registration at the binder's end:

100 - New method is registered
101 - The same server/port has previously registered for this function
102 - A new server has registered for a function that already has other servers serving it

The binder and server are multithread and spawn a new thread every time a new message is received. However, when the binder spawns a thread with a connection to a server, it keeps that thread alive even after its done its done work. This is useful to keep a track of servers when they die. This was achieved by having a send response code of 170 whenever the binder responds to a server. The return code on send told the thread to stay alive.

In most other places, I made do with -1 and 0 for error codes and signals. I used the response code of 2 when a client tries to allocate more memory than the server wants to (from the discussion groups I gathered that the server can have a soft upper bound on how much memory it is prepared to allocate to serve certain functions with array arguments)

Incomplete functionality

The second part of the rpcCall - to execute the actual function is incomplete.