CS454

Assignment 3

Design by: Johnson Chan and Andrew Jenkins

System Manual

Marshalling/Unmarshalling Data

The protocol for sending and receiving data between clients, server, and binder is as follows: first send an integer representing the length of a message to follow, then send an integer representing the type of request, then proceed to send the proceeding messages, which depend on the request/response type.

The length message is used to receive 'x' number of arguments for a function in the form of 'x' messages. From the request type, the binder or RPC library will decipher, via hard-coded switches, the number of messages to follow and their argument types. In other words, data marshalling and unmarshalling is hard-coded and dependent on the type of request.

rpcInit: sends/receives no more data than initial length and type messages.

rpcRegister: sends: register request (int), hostname (char\*), port (int), function name (char\*),

argTypes (int\*)

receives: response code (int)

rpcCall: sends: location request (int), function name (char\*), argTypes (int\*)

receives: function location response (int), port (int), hostname (char\*),

sends: execute request (int), function name (char\*), argTypes (int\*), arg1 (void\*),

arg2 (void\*) … argn (void\*)

receives: response code (int), function name (char\*), argTypes (int\*), arg1 (void\*),

arg2 (void\*) … argn (void\*)

rpcExecute: receives: function name (char\*), argTypes (int\*), arg1 (void\*), arg2 (void\*)

… argn (void\*)

sends: response code (int), function name (char\*), argTypes (int\*), arg1 (void\*),

arg2 (void\*) … argn (void\*)

Binder Database

The binder database is in the form of a map of server\_info and vector<function\_info> where:

- **server\_info** is a struct containing server\_socket (int), server\_name (string), port\_num (int)

- **function\_info** is a struct containing function\_name (string), argTypes (int\*), numArgs (int)

Insert a diagram here to describe the database?

Insertion of a server into the binder database happens when rpcRegister is called by a server. When the binder receives the register request, handle\_register\_request is called.

First, the binder checks whether the server or the function is already in the database. If the server and the function are both already in the database, a DUPLICATE\_REGISTER message is sent back to the server.

If the server is not in the database, we create a server\_info struct containing the server's information which was sent in the register request, then insert the server to the map, adding to the list of function\_infos the information for the function sent in the register request.

If the server is already in the database, but the function is not in the server's list of function\_infos, then a function\_info struct is created containing the information sent in the request, and added to the back of the list in the list for the server within the map.

A success message is then returned to the server, to signify that the function has been successfully registered (and that the server has been registered, if it is new).

Function Overloading

Scheduling

We use a vector vector<server\_info> called all\_servers to store all the connected servers in order to accomplish round robin scheduling, where:

- **server\_info** is a struct containing server\_socket (int), server\_name (string), port\_num (int)

When a client makes a rpcCall to a specific function, the binder searches through the binder database of all servers for every server that has said function registered—creating an array called valid\_server containing these servers in the order that they are found. Then the binder will search through all\_servers from the beginning to end to see if the server is in the valid\_server.

Following this, the first server in the list is sent an rpcExecute message for the function that is called via rpcCall from the client. In the server database, the server that was called to execute the function is placed in the server database where the server in the end of the array was placed, and every server is moved one over to the left in the server database. This way, the next time rpcCall is called, every other server containing the function will be found before the server that last executed the function.

Following this, the first “valid server” in all\_servers is sent to the client. After sending the server location, binder will put the first “valid server” at the back of all\_servers so that next time it will have lower priority being called.

Insert a diagram here to describe round robin scheduling?

Therefore, the system implements a form of round-robin scheduling for remote procedure calls.

Termination Procedure

We use a vector vector<int> called servers\_socket to store all the connected socket of servers.

When the binder receives a TERMINATE request, it proceeds through the list of server sockets and sends a TERMINATE request to every server individually.

If a server is in the middle of execution, and receives a TERMINATE request, the server will wait for all threads to finish execution (wait until number of threads < 0 and do not allow creation of any new threads), then stop listening for requests, and cease execution.

Once the server is closed, the binder will remove it from binder database, all\_servers list and servers\_socket list. If all these three data structure is empty and the TERMINATE requests is sent, binder will then terminate itself.