Data Transferring (Marshalling)

We created some helper functions in the file named "message.h" to help send and receive messages.

- For **strings**, first send/receive (1+length of the string), and then send/receive the string itself if successful.
- For **integers**, first send/receive the integer size, and then send/receive the integer itself if successful.
- For **integer arrays** (terminated by "0"), keep sending/receiving each array element until 0 is reached
- For **arguments (void*) array**, we gain information of the number, type, and length of the elements to be sent/received via another input to the function, argTypes. We memory-copy the element into the buffer, and then keep sending until we have sent all of them. For receiving, we keep receiving the elements into the buffer, and memory copy the message into our variable.

With the help of these functions, string messages (host names, functions names, request types, and response message), integer messages (port number, error code and warning), integer array messages (argument types) and void* argument messages (args) can be directly sent and received.

Structures Description

Struct Function: store the function's signature, including name, argTypes, and key

- "**name**" and "**argTypes**": received when some server tries to register the function
- "**key**": a string concatenated with function signature information. Key is of the following format:
    - Function name
    - For every element in argTypes, we only distinguish between scalar and array argument.
        - Scalar: if the last 16 bits of argTypes[i] is 0, concatenate with argTypes[i]
        - Array: if the last 16 bits of argTypes[i] is greater than 0, we only have to record the length is non-zero. So we concatenate with ((argTypes[i] & 0xFFFF0000) | 1).

Struct Server: store the server's information, including **host** name, **port** number, and socket file descriptor **fd**

Binder Database

First of all, in order to optimize performance so that each registered server's information is recorded only once, we created a vector of structure **server**'s, vector<server*> called **serverVector**, in server's registration order.

Then in binder, we used a map<function, vector<int>>, **functionMap**, to build the database.

- Key: **function** structure, as described above
- Value: a vector of server indices in **serverVector**. Those servers provides this function

When a server tries to register a function, our mechanism does the following:

1. First in **serverVector**, check if the server has already been registered.
    - If server has been registered, then find the index of this server record in **serverVector**.
    - Else, push the server at the end of the **serverVector**, and remember the index
2. Then in **functionMap**, find if the function has already been registered. i.e. find
    - If function has already been registered, then
        - If the server has already been registered with this particular function, then send warning
        - Else, push the server index in to the end of vector, (**functionMap**[function])
    - Else, add a new key-value pair, i.e. (function, server index) pair, into the map.

Server Side Database

We declare our server side database as map <struct function, skeleton> **serverDB**. Struct function is what we described above. Skeleton is the pointer to the procedure function. Inside the function rpcRegister( ), server will first call binder to register the procedure (as described in the above session “Binder Database”). Then server will wait for binder's response. If server receives a success message from binder, then the server will update its database by "serverDB[function] = skeleton".

When server receives the location request from client, server can find the corresponding skeleton according to the function name and argTypes. Firstly, we create a temporary struct function by name and argTypes. Then server can call "skeleton f = serverDB[function]" to get the skeleton and execute the procedure.

Client Side Database

We declare our client side database as map <struct function, vector<server*>> **clientDB**. Struct function and server are what we described above. The vector<server*> is used to store the servers' information. Since the binder will return a list of server information to client, thus we use a vector to store the servers' information.

Client will get one server’s information at a time from the back of the vector by calling "server* tempServer = serverVector.back();". If that server is not working, client will delete it from the vector by calling "serverVector.pop_back();". Otherwise, client will keep sending requests to that server for the same procedure. Client will ask binder for servers' information if the vector is empty.

Function Overloading

One of the major advantages of such a database structure is the ease to handle function overloading.
Structure **function** provides a custom comparison operator so that only functions with the

- same **key** (an element of struct **function**)

will be considered the same.

To be specific, since we manually concatenate the function’s all signature information into the **key** (as described in “Structure Description”), if two keys are the same, then they have the

- same function name,
- same argTypes (every element from both sides is of the same type)
- same argTypes array length (both scalar or both array)

Therefore, only functions with exactly the same signature will be considered the same.

If different servers register functions with the same signature (same **key**), then we store the index of the server to the back of server-index vector, **functionMap**[function].

Round-Robin Scheduling

The file binder.cc maintains a round robin queue, **RRqueue**, to guarantee round-robin scheduling.

1. Every time a new server is registered with some function, add the server's index in **serverVector** to the back of **RRqueue**. Therefore, a first-in-first-out mechanism is provided.
2. Every time a client sends a server-location request regarding a function f, we go through the **RRqueue**, from beginning**,** to check if any server A provides such function request, i.e., check if A's index in **serverVector** is in **funtionMap**[f]. If found, we move A's index in **RRqueue** to the back of it. Otherwise, move to the next server B and check if B provides the service.

By doing so, A will serve the request f again only if all other registered servers providing f in **RRqueue** have severed the request.

Termination Procedure

After receiving "termination" request from client, binder will

1. Loop over **serverVector**, inform every registered server to terminate by sending "TERMINATE" message, and close their sockets.
2. Close its own socket so that no more connection will be accepted.
3. Do all the clean-up so that any memory allocated to **serverVector**, **functionMap** will be freed.

Error Code

rpc.cc	General - connect to binder - connect to server	0 : function implemented successfully -110 : ERROR: failed on getting hostname -111 : ERROR: failed on opening socket -112 : ERROR: failed on connecting
	rpcInit()	-100 : ERROR: failed to get machine hostname -101 : ERROR: server failed to open socket for client -102 : ERROR: server failed to bind -103 : ERROR: server failed to listen -104 : ERROR: fail on loading BINDER_ADDRESS -105 : ERROR: fail on loading BINDER_PORT
	rpcRegister()	-120 : ERROR: failed to get machine hostname -121 : ERROR: failed to get socket address
	rpcCall()	-140 : ERROR: fail on loading BINDER_ADDRESS -141 : ERROR: fail on loading BINDER_PORT
	rpcExecute()	-170 : ERROR: server doesn't have the asking procedure -171 : ERROR: execute failure in the procedure
	rpcTerminate()	-180 : ERROR: fail on loading BINDER_ADDRESS -181 : ERROR: fail on loading BINDER_PORT
	rpcCacheCall()	-145 : ERROR: fail on loading BINDER_ADDRESS -146 : ERROR: fail on loading BINDER_PORT
binder.cc:	General - initialize socket - print binder info - select server/client	0 : function implemented successfully -200 : ERROR: binder failed to open socket -201 : ERROR: binder failed to bind -202 : ERROR: binder failed to listen -203 : ERROR: binder failed to achieve machine name -204 : ERROR: binder failed to achieve socket address -210 : ERROR: binder failed to select -211 : ERROR: binder failed to accept
	Handle Register	230 : WARNING: binder has encountered previously registered procedure -220 : ERROR: binder encountered invalid argTypes
	Handle Loc_Request Handle Cache	-240 : ERROR: binder has not found any available and alive servers providing requested function
message.cc	Send and receive messages	-300 : ERROR: failed to send string message -301 : ERROR: failed to send integer message -302 : ERROR: failed to send array message -303 : ERROR: failed to send args message -305 : ERROR: failed to receive string message -306 : ERROR: failed to receive integer message -307 : ERROR: failed to receive array message -308 : ERROR: failed to receive args message

Functionality

We have implemented all required functionalities specified in the document, including the bonus part. We also handle the cases when some server is down before client request “terminate.”

How to test cases when some server is down before client request “terminate”:

Suppose the only server A has registered with function f before a client tries to call rpcCall().

a. Terminate server A and run client. We will have error code -240 indicating no available (alive) servers providing the service.
b. Terminate server A and run client. Before a client tries to call rpcCall(), server A registers the function f again. We will see function f to be implemented by A.

Suppose we have servers A and B registered with function f.

a. Terminate A and run client. We will have server B providing the service.

How to test rpcCacheCall:

Request code for client: “CACHE”

Case 1. A binder with a list of registered functions and servers.

A client calls rpcCacheCall with the same function (name and argTypes) several times to see if client asks binder for a list of servers at the first time.

a. Assume the servers will never crash. Client will not ask binder for servers' information again (no output on binder side).
b. Terminate the first server in the list. Client will use the second server in the list instead.
c. Terminate all of the servers in the list. Client will ask binder for the servers' information again. (we can see cache request message on binder side).
d. Terminate all of the servers in the list. Client will ask binder again. If binder return null, client will not ask binder again.

Case 2. A binder with no registered functions.

A client calls rpcCacheCall with failure response(no server). Client will not ask again.