Documentation and System Manual

rpc.cc:

the protocol we follow to send and receive message is:
message_length, message_type, message

the message_length and message_type represent in integers. and message is represented in char array.

in message, each element in message are separated by a space char. so it can be parse it later on the other side.

when send the message, the data type ordering is like this: 4 bytes char (message length), 4 bytes char (message length), char array (message, for example: [name space argtypes space args]). the recv side obey the same order in order to receive data.

binder.cc:

in binder, the data structure to store the server information is like below:

```
struct entry {
    string name;
    int* args;
    int args_length;
    vector<string> servers;
};
```

name is the function name, args is the list of args, args_length is the size of the args array. vector<string> servers is list of server to the function.

For handling the function overloading. we loop through the database (vector of struct entry) and check if the size of the argtypes is same as the argtypes in the database or one is scalar and the other is an array under the condition that both have the same function name. if that is true, then we add the server into the database. otherwise, we don't store the server into the database since there already exist another server which has the same functionality.

The sameFunction() below do the checks in binder, return 0 for indicate that the allow server to be added into the database, -1 otherwise.

```
int sameFunction(int* arg, int arg_length, int* arg2, int arg2_length) {
    if (arg_length != arg2_length) {
        return 0;
    }
```

```
    for (int i = 0; i < arg_length; i++) {
        int array_length = arg[i] & 0xFFFF;
        int rest_of_arg = arg[i] & 0xFFFF0000;
        int array2_length = arg2[i] & 0xFFFF;
        int rest_of_arg2 = arg2[i] & 0xFFFF0000;
        if (array_length > 0 && array2_length == 0) {
            //cout << "different lengths " << arg[i] << " " << arg2[i] << endl;
            return 0;
        }
    }
    return 1;
}
```

For managing the round-robin scheduling. we are using two database source. one is the vector<string> db1 that store the concatenated server name and it's port number that registered procedure on binder. the other one is the one we just saw above, the vector<entry> db2 that store the list of functions name associated with related servers. at first, our return index i is 0. each time we check whether the element with index i within db1 has the procedure in db2. if it has, then send the string of server and port number concatenated in db1 to client, and increment index i. otherwise, we increment index i and check the next element in db1 and so on. if we loop through all element in db1 but find there is no server have the procedure that client need. then we send LOC_FAILURE to client.

For terminate procedure, we have another database source which is vector<string> db3 that store the same data type as db1 above, except db3 stores all server's address and it's port that the server connected to the binder even if it register no procedures on binder at all. therefore, after binder receive the terminate message from the client. then the binder will loop through the db3 and send the TERMINATE to every single server in that database. so every server will be terminated at the end.

For the error codes we used:

#define REGISTER_FAILURE -202

#define LOC_NO_SERVER -301 // reasoncode for LOC_FAILURE

#define LOC_BINDER_FAILURE -303 // reasoncode for LOC_FAILURE

#define EXECUTE_NO_FUNCTION -401 // reasoncode for EXECUTE_FAILURE

#define EXECUTE_SERVER_FAILURE -402 // reasoncode for EXECUTE_FAILURE

#define NO_PROCEDURE -403 // server can't find related function in it's DB

#define CACHE_NO_SERVER -501 // reasoncode for LOC_FAILURE in rpcCachecall()

#define CACHE_BINDER_FAILURE -503 // can't connect to binder in rpcCachecall()

Functionality implementation:

All functionalities of this assignment have been implemented.