Assignment 3
CS454: Distributed Systems
Dingzhong Chen and Andrew Jenkins

System Manual

Preface

The following is a system manual for Waterloo's CS454: Distributed Systems assignment 3. The exact assignment specifications can be found at the following link.The assignment specifies that in the system there are three types of remote processes: any number of clients, any number of servers, and a binder. The clients make remote procedure calls (RPC) on the servers through the binder. There are a number of possible different implementations for the binder functions and RPC library functions within these system specifications. The implementations in our version of this distributed system will be explained thoroughly below.

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 the number of arguments for a function, so that we may loop through and receive one containing for individual function argument. 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)

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. It does this by looping through the database and checking if there's a server with the same name and same port number in the database. If there is, it checks if the function is in the database by searching the list of function information contained in the database for the server.

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. We then add the function information to the list of function_info for the server.

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

In the binder server database, registered functions are stored in a list for each individual server. The registered functions are considered new and added to the list if and only if there is no function with the same name, the same number of arguments, and the same argument types.

If rpcCall is called on a function, the rpc library will also search for the function based on the function name, number of arguments, and argument types. Thus, multiple functions may have the same name, so long as they have different arguments.

If a function with the same name, number of arguments, and argument types as another function is registered, a DUPLICATE_REGISTER function will be thrown to the server, indicating that the function already exists in the database.

Scheduling

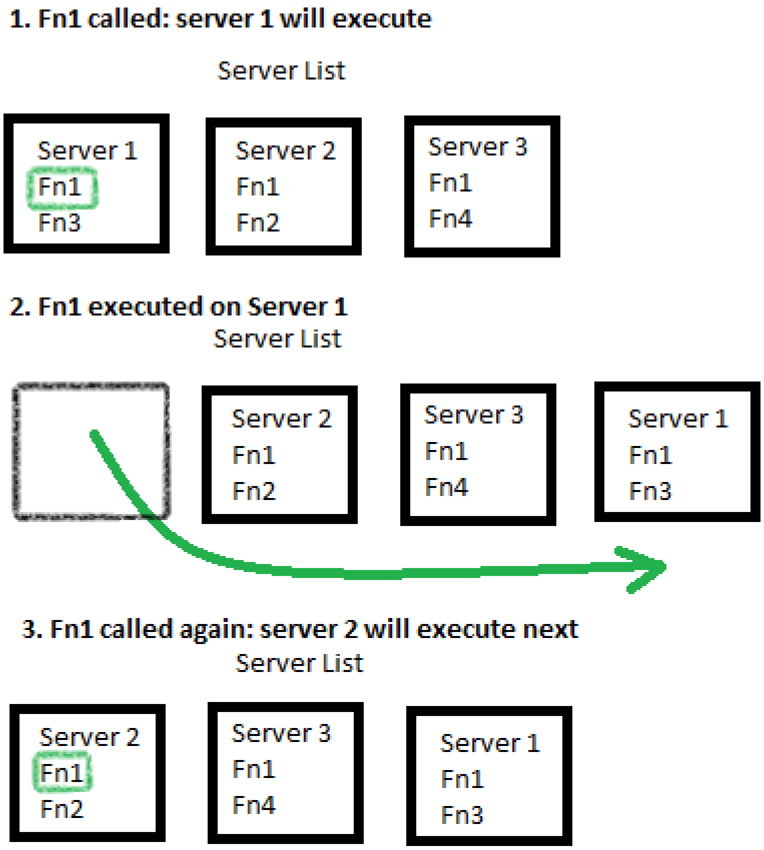
We use a vector vector<server_info> called all_servers to store all the connected servers in order to accomplish round robin scheduling, where:

- **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 the 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 "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.

Example:



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.

A server will first verify that the binder has sent the TERMINATE request. If a server is in the middle of execution, and receives a TERMINATE request, the server will wait for all threads to finish execution, then stop listening for requests, and cease execution.

Once the server is closed, the binder will remove it from the binder database, the all_servers list and the servers_socket list. If all three data structures are empty and the TERMINATE requests are sent, the binder will then terminate itself.

Error Code

Code	Meaning	Description
-1	REGISTER_FAILURE	Server is unable to register its procedure in binder
-2	LOC_FAILURE	Client is unable to get server’s location
-3	EXECUTE_FAILURE	Server is unable to run the procedure
-4	ADDRESS_ERROR	Server/Client is unable to get the address of binder
-5	PORT_ERROR	Server/Client is unable to get the port number of binder
-6	CONNECT_BINDER_ERROR	Server/Client is unable to connect to binder
-7	LISTEN_CLIENTS_SOCKET_ERROR	Server is unable to listen on client’s connection
-8	CONNECT_SERVER_ERROR	Client is unable to connect to server
-9	MSG_LENGTH_ERROR	Unable to get message length
-10	TCP_INIT_ERROR	Binder is unable to create a new socket
-11	BINDER_LISTEN_ERROR	Binder is unable to listen on connections
-12	BINDER_PORT_ERROR	Binder is unable to set port number
-13	GET_MSG_TYPE_ERROR	Unable to get message length
-322	TYPE_ERROR	If the type of message doesn't exist in constants

Unimplemented Features

None of the bonus functionality requested in the assignment specifications are implemented. That is, no cache system in the RPC library or binder were implemented.