CS654: Assignment 3

Done by: Shreya Agrawal 20503430, Ankita Dey 20546495

Due date: 19th March, 2014

1. Message Protocols

The message sending protocols are the same as suggested in the Assignment description and have hte following generic form:

LENGTH, TYPE, MESSAGE

This whole form is sent as an array of unsigned char (typedef as *byte*). The pointer to this array of *bytes* has been typedef as *message*.

The LENGTH is the total number of *bytes* of TYPE and MESSAGE.

The TYPE is any of the following:

REGISTER

REGISTER_FAILURE

LOC_REQUEST

LOC_SUCCESS

LOC_FAILURE

EXECUTE

EXECUTE_SUCCESS

EXECUTE_FAILURE

TERMINATE

MESSAGE_INVALID

SEND_AGAIN

LOC_CACHE_REQUEST

LOC_CACHE_SUCCESS

LOC_CACHE_FAILURE

The MESSAGE consists of any of the following as defined by the protocol defined in the assignment:

`server_identifier, port, name, argTypes, args, reasonCode`

a) Protocol for sending:

When an entity (client/server/binder) wants to send a message to another, it first calls a function `create<type>Msg` and passes the TYPE, and arguments of MESSAGE. As an example let's say we want to create a REGISTER message. The server calls `createRegMsg()` with arguments server_identifier, port, name and argTypes. For easy unmarshalling of message on the receiving side the following are stored as an array of bytes of fixed length given as follows:

LENGTH: sizeof(size_t) defined as DATALEN_SIZE

TYPE: `sizeof(int)` defined as TYPE_SIZE

server_indentifier: 32 defined as HOSTNAME_SIZE

port: `sizeof(int)` defined as PORT_SIZE

name: 16 defined as FUNCNAME_SIZE

reasonCode: `sizeof(int)`

This function returns the created final array of bytes pointed to by `message` and the server send it to the binder. The size of args is determined from argTypes argument. And marshalled in the message by copying the value stored at each pointer `args[i]` into an array of bytes of length of its data type.

b) Protocol for receiving:

An entity receives a message of unknown length by calling the function `recv()` and first receiving only DATALEN_SIZE number of bytes. On reading these bytes the receiver knows how many more bytes of the message to read and calls the function `recv()` to receive the rest of the message. This complete received `message` and its length is then passed to a function called `parseMsg(message msg, size_t length)` which does the work of unmarshalling the message. The unmarshalling is done by reading first TYPE_SIZE number of bytes, then HOSTNAME_SIZE and so on. The length of argTypes is determined by reading till the first `argTypes[i]` read is 0. The read arguments are stored in a `struct`.

2. Databases

a) Database at the binder:

The registered functions are stored at the binder using the `Vector` container of the Standard Template Library. The vector, called `serverStore`, contains elements of type `Server*`, where Server is a struct defined as follows:

```
struct Server {
     location *loc;
     std::set<skeleArgs*, cmp_skeleArgs> *functions; };
```

`location` is a struct that contains a pointer to the hostname and the port. It is used to identify location of the Server. `functions` is used to contain all the functions registered at this particular function. `skeleArgs` is a struct that is used to identify each function, i.e., its name and argTypes.

```
struct skeleArgs{
    char *name;
    int *argTypes; };
```

```
struct location{
    char *IP;
    int port; } ;
```

The use of the `Set` container allows us to handle registeration of functions with the same signature as an existing function since only unique functions can be stored in the set functions. The comparison function `cmp_skeleArgs` allows us to handle function overloading since it compares both the name and argument types. A different set associated with each different server allows us to register functions with the same signature but located at different servers.

The round-robin functionality at the binder for load balancing is achieved by searching for a matching function at a Server from the beginning of the vector till its end. If a matching function is found, that particular Server is removed from its currect position in the vector and pushed back in the vector. Since, this was the most recently served Server it will be the last to be found at the next call to `rpcCall().`

The server also contains a list of all servers that are currently up and connected to the binder. This is a list of all socket file descriptors open with the server and stored in the Set container. This list serves the purpose of sending the TERMINATE message to all servers when requested to do so by a client. When a server goes down it's sockfd is removed from this list and so is this Server removed from the Vector serverStore.

b) Database at the Server:

At each server, all its registered functions are stored in the `Map` container of the Standard Template Library. The Map takes a key_type `skeleArgs*` and value_type `skeleton.` That is, it maps each signature of a function to its skeleton. When a client requests for a function, the unique key for the function is signature is searched for and the value returned.

```
std::map<skeleArgs*, skeleton, cmp_skeleArgs> store;
```

c) Database at the Client:

In case `rpcCacheCall()` functionality is called, the location of the servers are stored at the client using the Vector container of the Standard Template Library together with the function names and their argTypes for each server that the client sends request to. The vector, called serverStore, contains elements of type Server*, where Server, location and skeleArgs structure have the same definition as used in binder. rpcCacheCall proceeds similarly as rpcCall, with the difference being that a message querying the binder for server's location i.e. createLocReqMsg will be sent only if it cannot successfully locate a server in the Server store having the same function signatures. The name and argTypes parameters are passed to retrieveFromCache() which will search in the available list of servers for the given name and argTypes and returns the location details if it finds a hit. In case, there is no such function signature registered, i.e. if the client wants to execute a particular function for the first time, then a location request message is sent across to the binder. When the binder replies with a LOC_CACHE_SUCCESS message, then the details are first stored in the serverStore by calling insertIntoCache() which will now append a new row if it is a new server location or a new function signature for an existing server.

In both the cases where the rpcCacheCall() successfully locates a server location or sends across LOC_CACHE_REQUEST to the binder, it will subsequently send a createExeSucMsg() to the server by calling sendExecuteToServer() . So, from here, the rpcCacheCall() proceeds as the rpcCall().

3. Termination procedure

When a client sends a TERMINATE message to the binder. The binder sends the TERMINATE message to all the servers currently up and registered at the server using the socket descriptor on which the server connection was accepted. The servers receive the message on the socket descriptor which was used to connect to the binder. On receiving this message, the servers, kill all the threads serving any clients closes its socket it was listening on and then exits. Once all the servers close their connection with the binder, the binder itself exits.

4. Error Codes

The following error codes are returned in case of an error:

CONNECTION_CLOSED = -4

- If the connection is closed.

BINDER_NOT_FOUND = -3

- If the connection to the binder fails because the binder goes down or was not found.

SERVER_NOT_FOUND = -2

- If the connection to the server fails because the server goes down or was not found.

FUNC_NOT_FOUND = -1

– If the function was not found either registered at the binder or at the server.

OK = 0

- When there was not error encountered.

FUNC_EXISTS = 1

- When function to be registered already exists and old one is replaced by the new one.

SOCKET_CLOSED = 2

- If the socket being used for accepting or connecting is closed.

INVALID_ARGS = 3

- If invalid arguments were sent.

WARNING = 4

- In case of any warning.