# CS454 Assignment 3: Documentation and System Manual

## Group Members

| **Name** | **User ID** |
|---|---|
| Ah Hoe Lai | ahlai |
| Yifan Dai | y25dai |

## Marshalling/Unmarshalling of Data

We have a **segment** class, which performs marshalling/unmarshalling of the message length and type to/from a buffer as well as sending/receiving the buffer. This class is a simple representation of an actual TCP segment. Pictorially, it looks as follows:

<table>
<tr><td colspan="4">Message Length</td><td colspan="4">Message Type</td><td colspan="3">Message Information</td></tr>
<tr><td colspan="4">4 Bytes</td><td colspan="4">4 Bytes</td><td colspan="3">X Bytes</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>…...</td><td></td></tr>
</table>

We also have a **message** base class and ten derived classes corresponding to the different types of messages, which performs marshalling/unmarshalling of the message information to/from a buffer as well as sending/receiving the buffer.

Finally, we defined fixed lengths for the different types of data that we marshall/unmarshall to/from a buffer. They are defined as follows:

| **Type of Data** | **Lengths** |
|---|---|
| Server Identifier | MAXHOSTNAMELEN (i.e.: 256 Bytes) |
| Port | 4 Bytes |
| Name | 64 Bytes |
| Reason Code | 4 Bytes |
| Argument Type (i.e.: argTypes element) | 4 Bytes |
| *args* | |
| Character Argument | 1 Byte |

| | |
|---|---|
| Short Argument | 2 Bytes |
| Integer Argument | 4 Bytes |
| Long Argument | 8 Bytes |
| Double Argument | 8 Bytes |
| Float Argument | 4 Bytes |

In order to send a message, the segment class sends the message length and the message type first. The appropriate message class calculates the buffer length and allocates space for the buffer. The message information from the class members is copied into the buffer starting with data that has a fixed length (i.e.: server identifier) and ending with data that has a variable length (i.e.: args). Finally, we send the buffer using a socket.

In order to receive a message, the segment class receives the message length and the message type first. The appropriate message class allocates space for the buffer using the message length received. It then receives the message information into the buffer. A new message object is created using the message type received and class members are set using data from the buffer.

## Binder Database

In the binder program we handle the storing of information sent to the binder from the server. The binder program also handles sending the correct server information to the client at the clients request.

We have three main data structures in our binder for handling storage and three types of objects.The first type of object is ***server_info***, which is an object that holds details about a certain server such as, the server identifier (server address), the port and the socket. The second type of object is ***procedure_signature***, which is an object that holds details about a certain type of procedure, this object holds both the name of the procedure and the types of arguments it accepts (arg-types). The third type of object is: ***server_function_info***, which is an object that acts as a container for the two objects described above, each ***server_function_into*** contains an ***server_info*** object and an **procedure_signature** object. The purpose of ***server_function_info*** is to act as a mapping between the two types of object. There will be a unique ***server_function_info*** object for every possible unique server and procedure relationship that is accepted by our distributed system.

In our binder database, we also have three different types of data structures to help us manage the data we receive from the server. The first type of data structures is called ***procLocDict*** (procedure location dictionary) and is a dictionary that maps an ***procedure_signature*** object to a ***list of server_info*** objects since the same procedure can exist on multiple servers. This data

structure helps us handle function overloading in our distributed system. The second type of data structure we have used is called ***roundRobinList*** and is a ***list of all the server_function_info*** objects we have. This data structure also helps us to manage the round robin aspect of the system. The final type of data structure is called ***serverList*** which is a ***list of all the server_info***. The purpose of this data structures is to just keep track of all servers that are connected to the binder. It is to be used later during termination.

## Function Overloading

Function overloading is handled by the ***procLocDict*** where when the binder is provided a new set of procedures. It will scan through the existing procedures in ***procLocDict*** to check whether or not the new procedure previously existed for the specified server in the database. If it does exist in the dictionary and the server exists in the corresponding list, nothing will happen. If it procedure does exist in the dictionary, however the server is not in the list, the server will then be added to the list. If the procedure does not exist in the dictionary, a new entry of the dictionary will be added with the procedure as the key and a new list with the server entry will be added as the value to the new procedure key into the dictionary.

Afterwards, if the procedure and location pair is new to the database, it will then be added to the end of the ***roundRobinList*** for future use.

## Round-Robin Scheduling

Round-Robin scheduling is handled in the binder with the help of the ***roundRobinList*** which is a list of ***server_function_info*** objects, in more simple terms ***roundRobinList*** will contain a copy of every procedure signature and the server it came from. (For example if server A had functions: f1, f2 f3, and server B had functions : f1, f2, f3, Then the ***roundRobinList*** will hold {A:f1}, {A:f2}, {A:f3}, {B:f1}, {B:f2}, {b:f3}).

How Round Robin Scheduling is enforced is in the scenario when the procedure signature f1 (with some arg types) is called, we will iterate through the list from the beginning to the end until we either reach the first occurrence of f1 or we have reached the end. If we indeed did find an occurrence of f1, we will keep track of the corresponding server of that procedure signature, and then handle the request with the selected procedure signature. If we did not find an occurrence of f1 then the binder shall send a ***LocRequestFailed*** message back to the client.

Afterwards, we will iterate through ***roundRobinList*** again from beginning to end and move every object that corresponds to server A to the end of the array. As a result functions of server A will be held at the “lowest priority” and will only be called upon if no other server holds the requested procedure in the following iteration. And it will eventually get pushed back up to the front of the list after several iterations of non use.

## Termination Procedure

Termination procedure is handled by multiple parts of the system. Firstly the client will trigger ***rpcTerminate*** which will send a TERMINATE call to the binder, after which the binder will handle this call by sending a single TERMINATE call to each of the servers it is connected to. The servers will handle the TERMINATE call by closing its sockets and then exiting its infinite loop. Afterwards the binder will close its own socket and exit its own infinite loop.

## Warning Codes

| **Name** | **Value** | **Description** |
|---|---|---|
| *Server Warning Codes* | | |
| WARNING_CODE_DUPLICATED_PROCEDURE | 1 | Is same procedure as some previously registered procedure |

## Error Codes

| **Name** | **Value** | **Description** |
|---|---|---|
| *Network Error Codes* | | |
| ERROR_CODE_SOCKET_CREATION_FAILED | -1 | Can't create a socket |
| ERROR_CODE_ADDR_INFO_NOT_FOUND | -2 | Can't get information about a host name and/or service |
| ERROR_CODE_SOCKET_BINDING_FAILED | -3 | Can't associate a socket with an IP address and port number |
| ERROR_CODE_SOCKET_LISTENING_FAILED | -4 | Can't tell a socket to listen for incoming connections |
| ERROR_CODE_SOCKET_ACCEPTING_FAILED | -5 | Can't accept an incoming connection on a listening socket |
| ERROR_CODE_SOCKET_CONNECTING_FAILED | -6 | Can't connect a socket to a server |
| ERROR_CODE_SOCKET_SENDING_FAILED | -7 | Can't send data out over a socket |
| ERROR_CODE_SOCKET_RECEIVING_FAILED | -8 | Can't receive data on a socket |
| ERROR_CODE_SOCKET_DESTRUCTION_FAILED | -9 | Can't destroy a socket |
| ERROR_CODE_HOST_ADDRESS_NOT_FOUND | -10 | Can't get the hostname of a host |
| ERROR_CODE_SOCKET_PORT_NOT_FOUND | -11 | Can't get the port to which a host socket is bound to |
| ERROR_CODE_BINDER_ADDRESS_NOT_FOUND | -12 | Can't get the hostname of the binder (i.e.: environment variable BINDER_ADDRESS was not set or incorrectly set) |
| ERROR_CODE_BINDER_PORT_NOT_FOUND | -13 | Can't get the port to which the binder socket is bound |

| | | |
|---|---|---|
| | | to (i.e.: environment variable BINDER_PORT was not set or incorrectly set) |
| *Server Error Codes* | | |
| ERROR_CODE_WELCOME_SOCKET_NOT_CREATED | -14 | Has not created a welcome socket to listen for incoming connections from the client (i.e.: rpcInit was not called before rpcExecute) |
| ERROR_CODE_NOT_CONNECTED_TO_BINDER | -15 | Is not connected to the binder (i.e.: rpcInit was not called before rpcExecute) |
| ERROR_CODE_NO_REGISTERED_PROCEDURES | -16 | Has no registered procedures to serve (i.e.: rpcRegister was not called before rpcExecute) |
| *Binder Error Codes* | | |
| ERROR_CODE_PROCEDURE_NOT_FOUND | -17 | Has no server that provided the desired procedure |
| ERROR_CODE_PROCEDURE_REGISTRATION_FAILED | -18 | Can't register the desired procedure |

## Unimplemented Features

**rpcCacheCall** has not been implemented.