THE AMERICAN UNIVERSITY IN CAIRO

الجامعة الأمريكية بالقاهرة

Distributed Systems

Fall 2019

Remote Invocation Middleware Project: Exercise #1

Group Members

Eslam A. Soliman 900163257

Fadi A. Dawoud 900163212

Ahmed Abouzaid 900163141

Mohamed Abdel Hamed 90016302

Build instructions:

1) The client

a) In the directory containing the source files, run the following command:

g++ -std=c++11 -o client.out ClientMain.cpp

b) The binary file 'client.out' will be generated. Then, run the following command:

./client.out <server IP address>

The client process will start and the server will be designated as the one at the given IP address

2) The server

a) In the directory containing the source files, run the following command:

g++ -std=c++11 -o server.out ServerMain.cpp

b) The binary file 'client.out' will be generated. Then, run the following command:

./server.out <server IP address>

The server process will start and will serve clients who designate it as server

Work distribution and meetings/decisions log.

Exercise 1:

Oct. 16th:

A group meeting to break down the requirement of the first exercise and the whole project's infrastructure into tasks and distribute them to the group.

Tasks assigning:

Message (Mohamed)

UDPSocket (Fadi)

UDPClientSocket (Ahmed)

UDPServerSocket (Eslam)

Client (Ahmed)

Server (Eslam)

ServerMain (Eslam)

ClientMain (Ahmed)

Bombarder/Jammer (Mohamed)

Discussion of components:

The UDPSocket class:

The UDPSocket class is the first level of abstraction above the Linux system calls provided by the Linux communication libraries. It is a wrapper around a socket descriptor and abstracts away the details of binding the socket to a socket address, designating self as a client or a server, and writing or reading from the socket with different options. Below are the implemented methods for this class:

1) UDPSocket(): The constructor of the class. It creates generates a new socket descriptor for the new object and sets the details to some default value.
2) (private) doBind(): it binds the socket to a local socket address, setting a flag that it is bound.
3) (private) initializeMyAddr(): it is a private method, whose function is to initialize the socket address object of the UDPSocket. Unless values are specified for the address and port number the socket should be bound to, default values are used (by passing 0 as port number and INADDR_ANY as an address).
4) initializeServer (char * _myAddr, int _myPort): this method designates the UDPSocket as a server with the given address and port number. It binds the socket to the provided socket address. It returns true if the process is successful.
5) initializeClient(char * _peerAddr, int _peerPort): this method designates the UDPSocket as a client of the server whose socket address is provided. If the socket is not already bound to a local socket address, it is bound with a socket address of default values. It returns true if the process is successful and false otherwise.
6) initializeClient(sockaddr_in client): same as above, but the socket address of the server is given as struct sockaddr_in object.
7) writeToSocket(char* buffer, int maxBytes): it writes up to *maxBytes* from the *buffer* pointer onto the socket. To invoke this, either one version of initializeClient() or readSocketXX() must be invoked (otherwise it will not do its function and will return -1). The recipient of the message is the latest to be designated with initializeClient() call or the latest to have sent a message received by the socket via readFromXX() call, whichever latest.
8) writeToSocketAndWait(char* buffer, int maxBytes, int microSec): same as above, but the socket waits for a total of *microSec* microseconds before returning.
9) readSocketWithBlock(char * buffer, int maxBytes): the socket blocks until it receives a message, storing in *buffer* a maximum of *maxBytes*. It returns the number of bytes received, and -1 on error.
10) readSocketWithTimeout(char* buffer, int maxBytes, int timeoutSec, int timeoutMilli): same as above, but it returns control after blocking for *timeoutSec* seconds and *timeoutMilli* milliseconds. It returns -1 on timeout.

11) readSocketWithNoBlock(char* buffer, int maxBytes): same as (9), but the socket does not block until it receives a message. If no message was found about invoking the method, it returns -1.
12) getMyPort(): return the port number the UDPSocket is bound to. If the socket is not yet bound, it returns -1.
13) getPeerPort(): number of the latest client/sender to the socket. If no such entity exists, it returns -1.
14) getSocketHandler(): returns the descriptor of the socket.
15) ~UDPSocket(): the destructor. It closes the socket.

The Message Class:

From an object-oriented point of view, a Message object represents the unit of communication between servers and clients conforming to the fundamental principle in distributed systems that communication is done primarily through message passing. For our class, there are two types of messages, namely, a Request and a Reply, Each message has an operation number, RPC identifier to clearly specify the function performed, and the remote procedure called. The Message class has the following methods implemented:

1) Message (int operation, void * p_message, size_t p_message_size,int p_rpc_id): The class constructor as perceived and used by the sender. It takes all the info needed for a message such as message data, rpc_id, etc.
2) Message (char * marshalled_base64): The class constructor as perceived and used by the receiver. It takes a data stream in Base64 format and unmarshals it to its original data contents.
3) marshal(): converts the message data into a Base64 encoded data stream encapsulating message parameters (e.g. message request, rpc_id, etc.) to be transmitted into the network. Base64 was preferred over other encoding systems like UTF-8 because it basically solves the problem of null characters so they don't get misinterpreted by the recipient. Also, they solve the problem of special (or control) characters altogether making no room for unexpected misinterpretations on the recipient side. However, of course, it comes at a cost of memory.
4) getOperation(): Returns message operation.
5) getRPCId(): Returns message remote procedure call ID.
6) getMessage(): Returns a pointer to a copy of the Message.
7) getMessageSize(): Returns message size.
8) getMessageType(): Returns message type.
9) setOperation (int _operation): Sets Message operation.
10) setMessage (void * msg, size_t msg_size): Sets Message contents (without marshalling).
11) setMessageType (MessageType msg_type): Sets type of Message (i.e. Request or Reply).

12) base64_encode (char* msg, size_t message_size): Static private helper method to encode messages in Base64 during the marshalling process.
13) base64_decode (char* encoded_string, size_t message_size): Static private helper method to decode marshalled messages in Base64 during the unmarshalling process.
14) ~Message(): Destroys the message and frees its memory.

Message Schema:

Message Type (1 Byte)	RPC_ID (1 Byte)	Operation (1 Byte)	Message Size (2 Bytes)	Marshalled Base64 Message (The remaining bytes)

The UDPServerSocket Class:

As the UDPServerSocket class is a derived class of the UDPSocket base class, we didn't find any need for any extra members in the derived class for the current exercise. Nonetheless, an empty .h and .cpp for the class are made for feature use. So, currently, the class is basically calling the base class constructor and destructor directly.

The Server Class:

Once an object is instantiated from the Server class, it uses a public member function to accept exactly one request and, for exercise 1, replies with the same message content as a message of type reply.

1) (private) Message* getRequest(): used by the class to block listen to its socket waiting for request type message. In the case of returning an error by the server socket or receiving a message that is not of type request, the function will return a null pointer.

2) (private) Message* doOperation(Message* _received): used by the class to pass a message of type request and returns a message depending on the operation requested by message "_received". For exercise 1, this function always returns a pointer to the same message except in the case of receiving a message with the string content "q", this will result in the termination of the server.

3) (private) void sendReply(Message* _message): a function used by the class to send reply message. It takes the output of doOperation and changes its type to reply then attempts to send it using the socket member of the server class. In the case of the socket write function returning an error code the function will report the existence of an error to the console.

4) Server(char* _listen_hostname, int _listen_port): the default and only constructor for the Server class, it takes the IP address with an agreed port number for the server to listen to, both parameters will be used on the client side.

5) serveRequest(): a void type function that is used to listen and serve exactly one message of type request.

6) bool isRunning(): a function that return a bool which is the state of the server, used to terminated the ServerMain depending on the values sent to the server.

7) ~Server(): the destructor of the object, it's only job, currently, is to delete the socket member of the class.

The ServerMain:

The server main is the main application for the server, it instantiates an object of type server and calls

The Client class:

The Client class encapsulates the client functionalities. It has an instance of the UDPClientSocket class, which implements the UDP client.

1) Client (char * _hostname, int _port): The client constructor receives the host name that the server process resides on as well as the port number.

2) Message * execute (Message * _message): The execute method receives a message which is transmitted for remote execution, and the reply message is returned.

3) ~Client(): Client destructor.

The UDPClientSocket Class:

This class represents a client UDP socket and it inherits from theUDPSocket class.

1) UDPClientSocket (): It has an initialization method that initializes the UDPSocket in a client mode.

2) ~UDPClientSocket (): Client socket destructor.

The ClientMain:

Prior to sending the request, the message type should be set as a “Request”. The type of the message received as a reply should be checked and its type should be a “Reply”. All the network operations details should be encapsulated inside the execute method, and the calling routine should not be aware of the remote execution.

The Client main has a jammer flag to activate or deactivate the Jamming process, which keeps sending requests to the server.

If the jammer flag is false, the client is able to enter a message continuously (up to 8192 char) to be executed and sent to the server with type “Request”, which will receive a reply from the server. Users can terminate the client and server by pressing “q”. After each loop of entering a message and receiving a reply from the server, both request and reply is deleted to free up the memory.