COMP3331 Computer Networks Assignment 1

Author : Jabez T Wilson (z5027406)
Date : 28/4/2017
Lab class: Fri 13-15

Overview

The objective of this assignment is to create a instant messaging platform while implementing out own application layer protocol. The programming language chosen to implement this was chosen to be C. since everything is a stream of chars (in TCP) it is important to recognize the data that comes in, so there are two data structures requestHeader and response. These act as headers for all messages between server and client and vice versa, the former for messages from client to server and the latter for messages from the server to the client. since this application layer protocol will be implemented over TCP, all data should be serialized/deserialized to a stream of chars. Another design implementation fact is that threading had to be implemented in both the server and the client. All of this will be detailed in the next sections.

Headers and other structures

Information about data definitions is held in the shared.h file. This file is shared between client and server. Other than data definitions it holds numerous MACROS used to denote error values, request values, message types etc. It also has function definitions for serialization and deserialization (explained in the next section)

requestHeader

the first attribute of this structure is secretKey, the value held by secretKey is the same for all objects. This is to verify that the data coming to the server has not been corrupted in any way. It also has an attribute telling the server what the client is requesting. The remaining attributes holds information about what data is following this header(messageType) and the length(msgLength) of it.

response

just as in requestHeader the first attribute in this structure is secretKey for the same reasons as above. It holds attributes for error values and what the response is for, this becomes important when the client has to deal with a sudden message from the server when it was not expecting it(when a message from a user is received). it's got attributes to tell the client what the data after the header is as well and finally it has an attribute that holds the duration of the time the user is blocked when they failed to login (this could be improved, refer second to last section)

key, keyValue and RAW

This a type of the data sent after the header (it’s messageType attribute would be KEY and KEY_VALUE). key is a struct with just a string(fixed length) attribute called key, and keyValue is a struct with two string(fixed length) attributes called key and value. RAW messageType means that the rest of the data is a null terminated string.

The application is not forced to send only one of these structures with every header, multiple instances of the SAME type can be sent with a single header instance as long as the msgLength property is set appropriately.

Another type of data that can be sent has messageType value KEY_AND_RAW. If an application reveived this messageType it means that the first n(size of the structure key) bytes belongs to a key instance and the rest is a simple string.

Serialization and Deserialization

Since TCP accepts a string of bytes (chars) all data structures should be serialized and deserialized at the other end. To do this serialization and deserialization functions are written in the shared.h file. They mostly follow the same call signatures(string is an exception, it needs the length as well)

> *char* serialize_<data type>(char* buffer, <data type> a)*
>
> *char* deserialize_<data_type>(char* buffer, <data type>* a)*

the first argument is the buffer to read to or write to, the second is either the data type or a pointer to it(if writing to it). The return value is points directly after the data_type in the buffer. For example if serializing an int(4 bytes) the return value will be buffer+4, same for deserialization. This makes it easier to chain multiple serialize/ deserialize functions. Infact all the user_defined data structure serialization/deserialization functions call the same functions for each of it’s attributes.

Multithreading

The server should be listening to the ports it already knows as well listen for new ports, this functionality had to be implemented with threads. The main thread will parse through a list of known ports and await any command from it or close it due to inactivity, while another worker thread accepts new connections at the welcomeSocket and adds it to the list of known sockets. To avoid data corruption mutexes were employed to make sure both threads did not try to use the list of known ports at the same time.

Similarly in the client, the client should wait on the user as well as handle sudden messages from the server(user messages). The main thread waits on the user and the worker thread listens for messages. The worker will listen only if the main thread is not already actively writing to or reading from the socket. A mutex is used to make certain the socket is not simultaneously being used.

Improvements and shortcomings

A number of improvements can be made to this project. One of them is the duration attribute on the response object. Since only a (failed) login attempt needs to know this value, 4bytes of data is wasted everytime the server sends a response to the client. A more suitable place for this value would be in the message body itself maybe as a key structure.

IPs that try to sign in with a user that is already signed in is not being blacklisted as specified in the assignment spec.

A checksum was not implemented though functions and data types to aid in this were implemented. A simple approach would be to add a 4 byte(int) footer at the end of every message (inclusive of header).

References

http://stackoverflow.com/questions/1577161/passing-a-structure-through-sockets-in-c, user unwind provides inspiration for the serialization/deserialization code

http://man7.org/linux/man-pages/dir_all_alphabetic.html, The linux man pages

https://www.gnu.org/software/libc/manual/html_node/Waiting-for-I_002fO.html, Waiting for Input or Output, guide to using select to wait on a socket.