Sockets

Introduction.

This report covers the implementation of a simple chat consisting of a server and several clients. ChatClient.java: Has two threads, one to listen for incoming messages from the server and one to send messages to the server. ChatServer.java: Has one thread to each of the clients currently connected but also one thread to listen for new incoming connections from new clients.

Client.

ChatClient.java consist of two classes, the ChatClient class itself and ChatSend. In the main method we create a new socket to localhost port 1234 as well as taking the username as an argument for later use. The newly created socket is passed as an argument to a method called receive. In this method, we create a buffered reader with an input stream reader reading from the sockets input stream of data. this reader is continuously monitored for new incoming messages which are printed to standard output. In case of failing the exception is catched and is being printed as a stack trace.

Back in the main method a new thread is spawned to handle the sending of new messages. This thread is told to execute the ChatSend class. This class takes the same socket as receive as an argument but also the username that was previously saved. In order to for this class to run in a separate thread the class implements the Runnable interface. The class only has one method run() where a new print stream is created on the sockets output stream. In order to read input from the user a buffered reader is created on system.in.

In order to display usernames in the chat the method starts with sending the users username to the server. The method then enters a continuous loop where every new line inserted by the user is sent to the server. In case of failing the exception is catched and is being printed as a stack trace.

The client is now executing two threads, one that is responsible for listening for new messages and the other thread is responsible for sending new messages.

Server.

ChatServer.java consists of two classes, the ChatServer class itself and a ClientThread class. In the main method we create a serversocket on port 1234. An array list named connections is also instantiated in order to save all current connections. A simple message is printed to std out once the server is started displaying on which IP adress and port the server can be found. The main method is then entering a continuous while loop that is checking for incoming connections from new clients. Every incoming connection on port 1234 is accepted. A new object and thread is spawned for each new client connection in order to be able to handle a large number of simultaneous chatting clients. The new socket connection is passed to the newly created object for later use. The new connection is also appended to an unsorted array list called connections.

To keep the list of all connections updated across all clients the ClientThread class has a public method updateList() that is called before spawning the new thread in main. This method takes the new array list as an argument and replaces the old one with the new list.

Each newly created thread starts out in the run() method in the clientThread Class. Here, a buffered reader is created reading from the new clients sockets / connections input stream.

The next thing that is done is that the username that was sent from the client is received and saved in a string for later use. Additionally, the thread calls a method called newConnection with the newly received username. This method is responsible for broadcasting to all other currently connected clients that the new user has joined the chat. This is done be utilizing the array list of connections passed from the main method in the ChatServer class. The method is simply iterating the list and sending an informative message to all clients in the list.

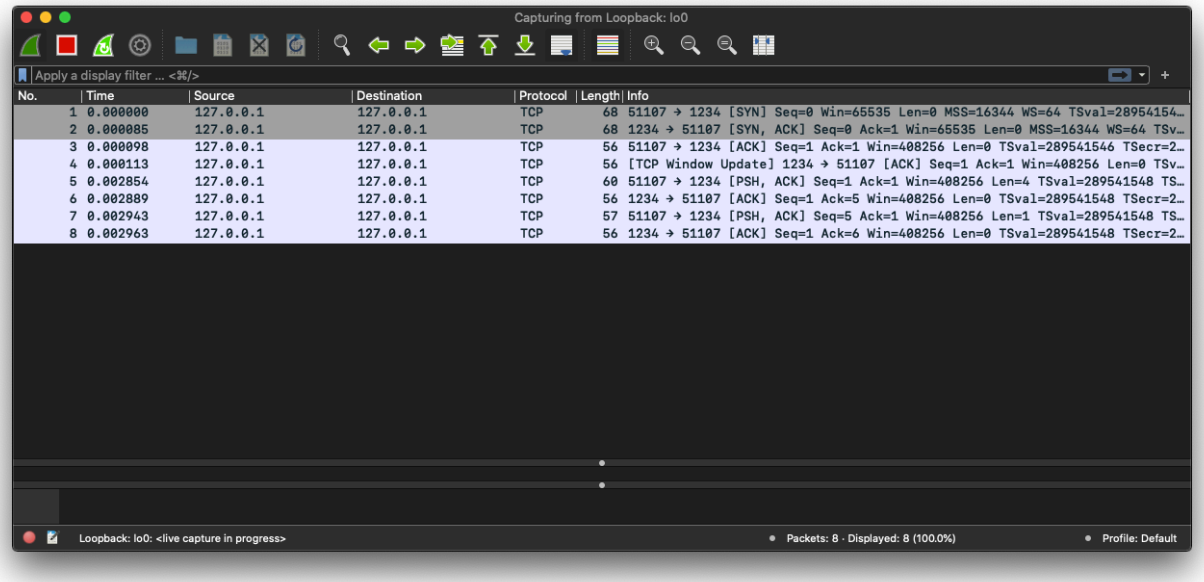
After this, back in the run method it enters a continuous while loop checking for any incoming message from the client, when a new message is received it is being passed to the broadcast method. This method is implemented in a similar manner as newConnection, i.e. broadcasting the message to all clients in the list. The message received from the client is sent to each connected client (except the client that sent the message) with the username prepended in order for all clients to distinguish from which client the message was received. In case of failing in any of the methods the exception is catched and is being printed as a stack trace.

The server is now executing one thread listening for incoming connections as well as one thread per connected client. This makes it possible for the server to handle a large number of simultaneous connected clients.

Extra assignment.

In order to capture network traffic from the chat we use the network sniffer Wireshark. Sniffing is done on the loopback Io0 interface since we are connected to localhost on 127.0.0.1.

Connecting one client to server

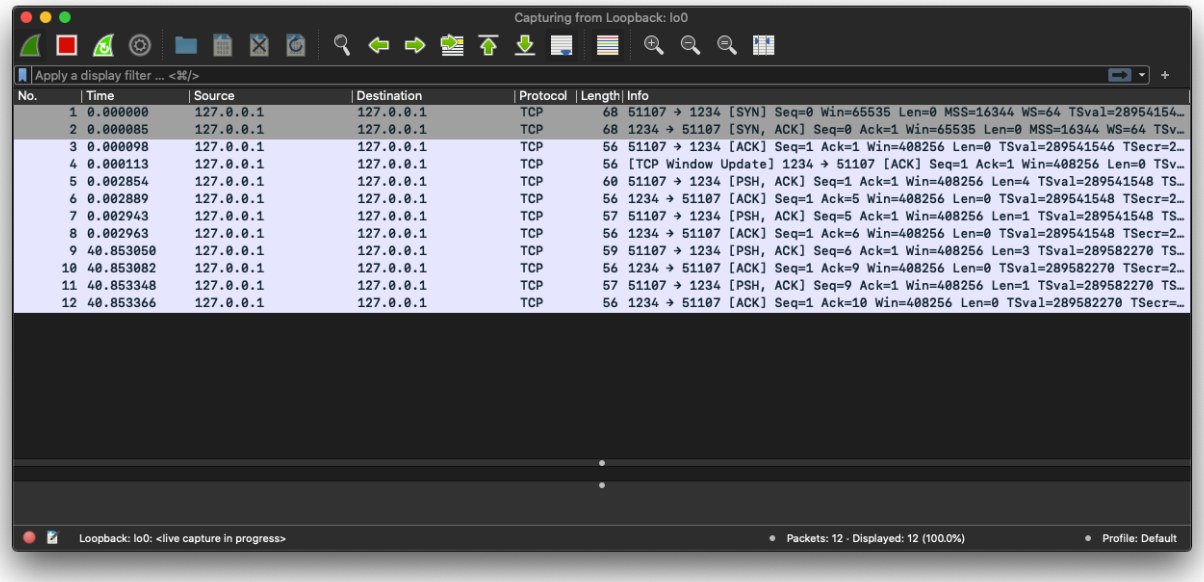


Here we can see that the first three rows is the standard TCP three-way handshake. After that the window size is updated. Row 5-8 is showing the network traffic of the client sending its name to the server and corresponding acknowledgements. In this case its sending ”Emil” which is len=4. A note

on the PSH flag, PSH in the TCP header informs the receiving host that the data should be pushed up to the receiving application immediately.

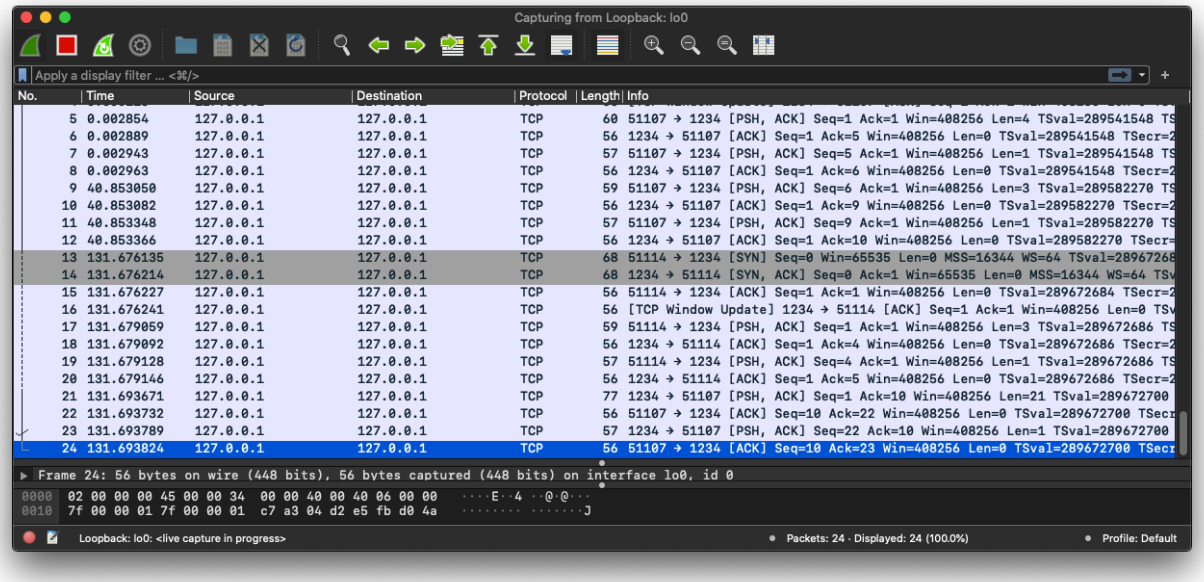
Sending one message from client to server

Here we can see that sending one message to the server results in four rows in our packet sniffer. Row 9 is the message itself while row 10 is the ack of the message.

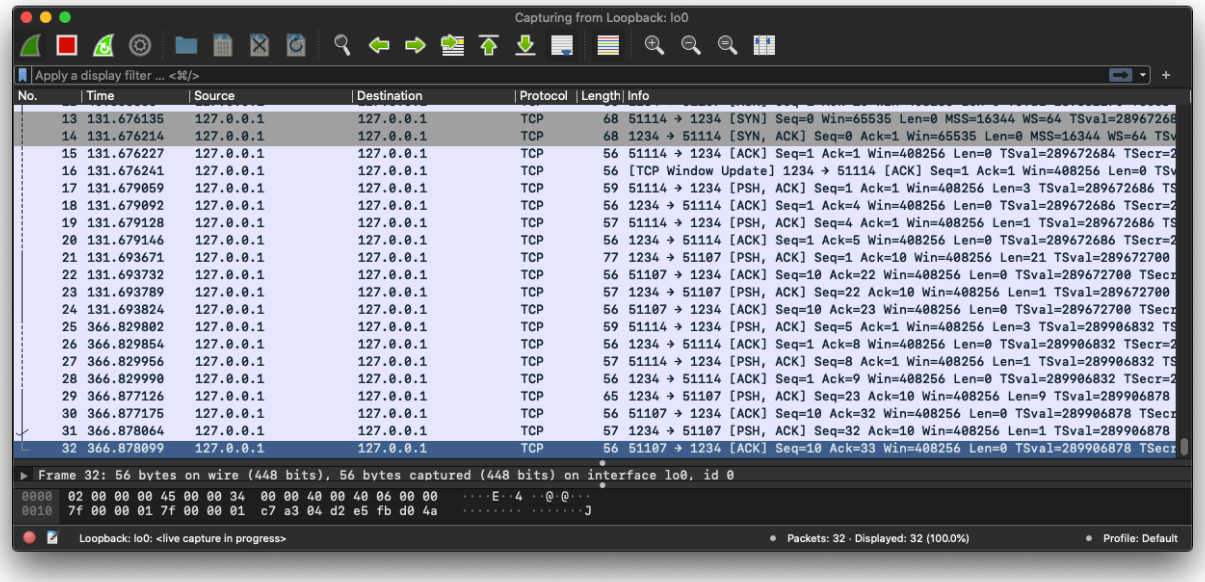


Connecting another client to server. Total of 2 clients.

By connecting another client we see the same TCP three-way handshake at row 13-15. After that the client is sending its username and the program is broadcasting that the new user has joined the chat. It is logical that this results in 8 rows since it previously was 4 rows with one client. The informative message can be found by expanding each row with a PSH flag.



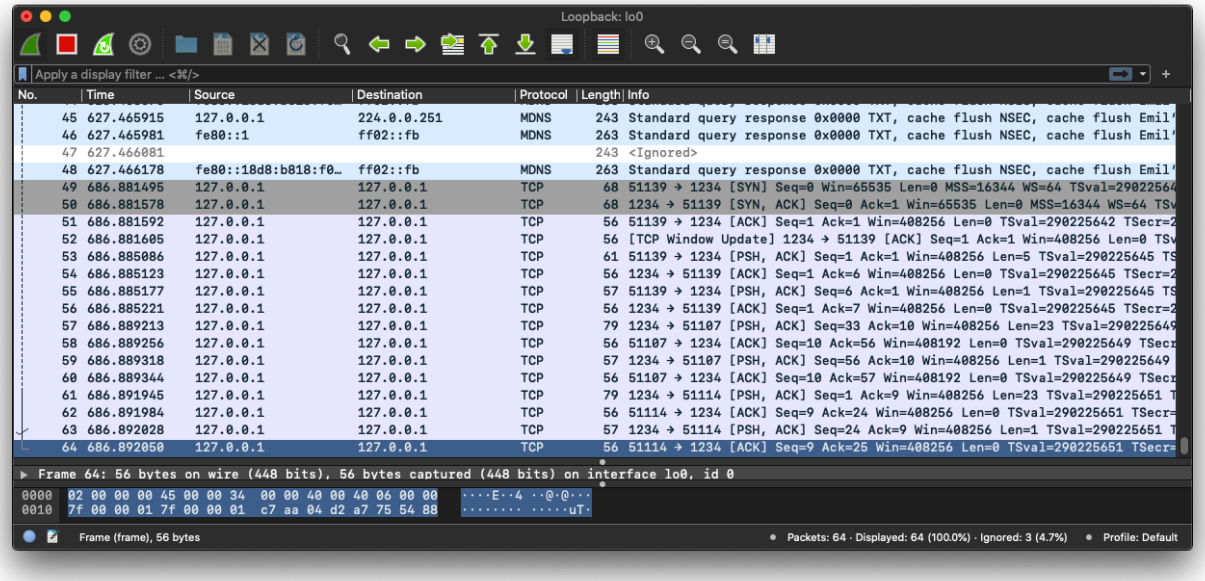
Sending one message to server with two connected clients



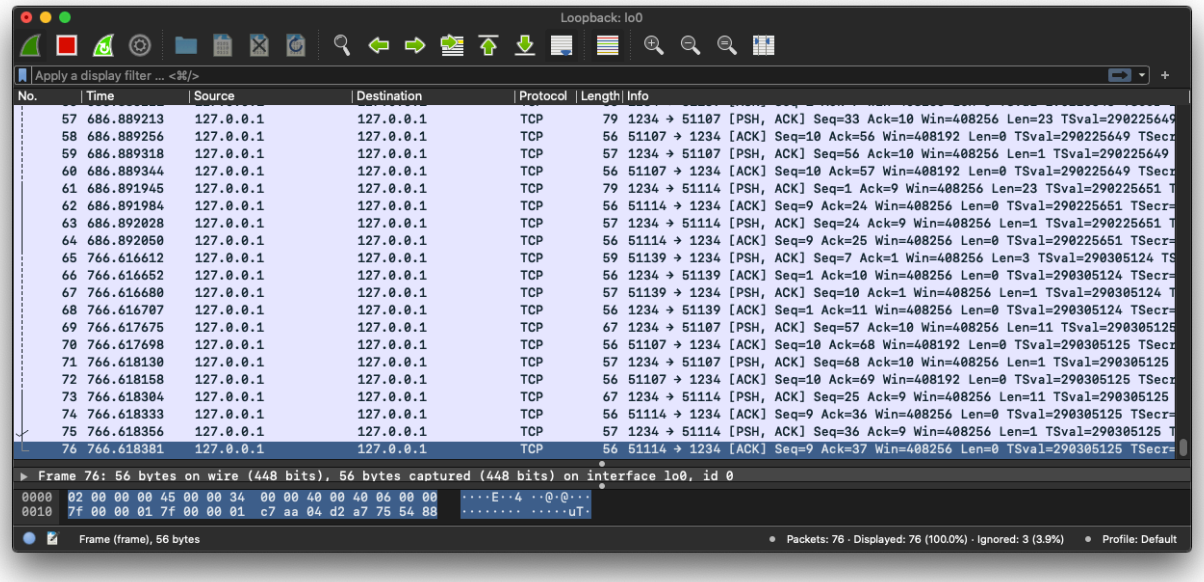
Here we see that sending one message with two connected clients results in 8 messages. Which is reasonable since it was 4 rows with only one client.

Connecting third client to server

When connecting a third client it results in the usual handshake of three rows and 12 rows for the informative message that a new user has connected to the chat.



Message from third client to server



Here we see that sending this message results in 12 rows, which is logical since each new user results in 4 more rows.

Difference between chat and web browser traffic

The most obvious difference between these are that the web browser sends a lot more and different headers and flags than we get in our chat. The browser is also using TLS (transport layer security) which the chat is not using in this implementation. Furthermore, the web browser does not close its connection after every bit of data, i.e. it is using HTTP Keep Alive where it checks the connection/ socket periodically (for incoming HTTP requests. The browser is also varying its window since while the chat keeps it the same. The chat is very sequential in its use of acknowledgements while the browser acknowledges in groups of 2-3. We can also see some DNS lookups in the sniffing results. We can also see some lesser known protocols like SSDP and ICMP and some local traffic that uses MDNS and ARP.