A4: Computer Networking (II)

Dea Felicia (dlr211), Said Said (hdn456), Kristian Jelsted (zxw463)

September 2023

This submission is the work of the indicated group members, and only the indicated group members. If ChatGPT, or an equivelant system, has been used at any stage of the report writing or code construction it has been clearly indicated.

1 Introduction

We have been asked to finish an implementation of a client for a file transferring service that uses peer to peer architecture. T do this we have to use socket programming and multi threading in C. We have also been asked to answer a series of theoretical questions concerning TCP.

2 Theoretical question

1.1.1

part1: The three-way handshake is important because both sides needs to acknowledge each others sequence number, before they can proceed. The first party sends their sequence number, the second party acknowledge it and sends their own, the first party acknowledge this.

part 2: TCP facilitate a full-duplex connection by only having a connection between a single sender and a single receiver where data can floe from a to b, while data flows from b to a

1.1.2

part 1: TCP is more reliable than UDP, but this comes at a cost. Because TCP has a bunch of reliabilty mechanisms and the 3-way hand shake, it also has a bigger overhead than ¡UDP, which is less reliable but also a more light weight protocol

part 2 TCP is reliable because it uses 3 way handshake, sequence numbers and flow control. This means it can guaranty that the packages are sent correctly, in order and without compromising the integrity of the data. Of course no a 100% guaranty exists, but TCP uses measures to ensure best reliability.

1.2.1

part 1:
1.1: RST is reset and is used to signify that a connection should be terminated immediately usually because of a communication error. RST can for instance be activated if you are trying to connect to a port that does not exist
1.2: The sequence number is the number of the first byte of the segment the byte stream is currently sending. The acknowledgement number is the number of the next expected byte segment the receiving host is expecting to get. The client and server have their own acknowledgement and sequence numbers that they each keep track off
1.3: The receive window is used to get an idea of how much free buffer space is left at the receiving host. If rwnd number is positive it means that there is more space left.

1.4: If the window size of the TCP receiver is 0, the sender is informed that no more free buffer space is currently available. When this happens the sender is required to keep sending one data byte segments until the receiver empties their buffer and starts sending non zero values again.

part 2: When the client sends the initial SYN to the server it sends it's sequence number and when the server sends the SYN/ACK back, it both acknowledges the clients sequence number and sends it's own sequence number, which technically makes it possible for the client to start transferring data since it now knows it has made connection to the server and what the sequence number of the server is.

1.2.2:

part 1: Congestion control in TCP usually uses AIMD(Additive Increase, Multiplicative Decrease). AIMD controls the growth of the congestion window by increasing cwnd linearly and then decreasing it multiplicative, by cutting it down in half. This also insures fairness, since cutting the cwnd in half allows other clients to use the network. AIMD is an end-to-end approach rather than purely network assisted

part 2: To determine the transmission rate, TCP uses the RTT and the Cwnd. The transmission rate is w/RTT, where w is the window size in bytes. The transmission rate is between $w \cdot /(2 \cdot RTT)$ and w/RTT which gives us the averge model of $\frac{0{,}75 \cdot W}{RTT}$

part 3: TCP sender infers the value of the congestion window through different tools. cwnd, to keep track of the rate at which TCP can send traffic. Slow start, Congestion Avoidance and fast recovery, which modifies the cwnd rate. Slow start makes it so it starts small and then increase exponentially. Congestion avoidance changes the cwnd according to whether or not congestion might be around the corner, so that id congestion might happen it decresees the cwnd. Fast recovery which is activated when a segment is missing and increase cwnd based on how many duplicate ACK is received for the missing segments.

part 4 The sender needs to have implemented a detection of ACKs to know if three duplicate ACKs have been sent and something that triggers a transmission of the lost segment. The receiver need to implement a duplicate ACK sender, so it can send multiple duplicate ACK. The sender should receive 3 duplicate ACK of the previous packet

3 Protocol and security

The design uses a custom protocol, where each peer registers using sockets. In order to transfer files according to a peer-to-peer architecture, peers manage their own network, enabling them to connect and retrieve files. Since the peers manage their own network, they communicate by sockets. Multi threading is used to allow concurrent client and server interactions.

We made sure to follow the protocol when receiving, replying between the peers. It reminds a lot of the last assignment, but making sure that we used the correct: client server responses and client peer request, would ensure that the peer to peer network between our own peers and the python peers would work correctly without issues.

To avoid race conditions we made sure that globally shared resources, in our case the network, which is both used by the client and server part of the peer, was under mutex lock and unlock. This means only one thread can access these shared resources at a time, which ensures mutual exclusion and prevents race condition of the shared resources.

To avoid deadlocks- mutex locks are utilized to prevent multiple threads from modifying functions. We also made sure that the mutex lock and unlocks of these shared variables, wouldn't end up in a deadlock, where one thread would wait for the other thread to unlock a globally shared resource, and it would never unlock. So there shouldn't be any shared variables that gets locked without never getting unlocked.

To ensure security and some form for validation the code hashes the file and the payload of the blocks in block_hash and total_hash, ensuring what we receive and send are the exact same files.

The code doesn't verify if the peer requesting the file is actually registered, which implies a security fault. This might be something to be improved upon in future iterations of the protocol.

4 Implementation and design

In our implementation of the peer to peer C client we have finished the requested functions and added a couple of helping functions of our own. We will go through them from the top as well as pointing out any changes made to already existing functions.

The first function we have altered is **send_message**. We have altered the 'Construct a request message' part of the code so it distinguishes between the commands retrieve, request and inform. We have then finished the implementation of the 'reply from server' part of send_message, to do this we have created a helping function.

The helping function is **handle_reply_fromserver**. In this function we extract the address information from the server reply, we then reallocate more space for the network and for the address. We have error handling in case our network variable is equal to NULL, meaning the allocation has failed. If this isn't the case, we copy the new address into an address struct where we have converted the port to network byte order. We then add the new address to the network array and print out all the network addresses.

The next function we have altered is **client_thread**. In this function we have added the functionality that the client continuously asks you to enter the name of the file you wish to retrieve or quit, if you would like to quit. If you enter the name of a file, we run the function get_random_peer to find a random peer and then use send_message to sent a message to the server about what to retrieve.

The next function we have created is **inform_peers**. In this function we use send_message and the network to inform all peers of a new peer. We start by the new peers ip and port into a request_body and converting the port to network byte order. We then go through all the networks and send an inform message to all peers that isn't the new peer. We use mutex lock when we access the networks since it's a shared resource.

Next we finished the implementation of the function **handle_register**, that handles requests to registers new users. When a new peer wants to join the network through another peer we use this function. We start by allocating space for the new address. We then go through all existing peers to see if the new peer already exist on the network. If the new peer already exists on the network we sent the reply header back to the server. If the peer doesn't already exist, we want to add them to the network. To do this we first reallocate the network space to allow for the new peer. We have error handling in case the allocation goes wrong. We then insert the new address into the network. Then we make a struct to hold the reply to the peer that registered with us and send the information to the server. Finally we use our inform_peer function to inform the other peers and then we print the network array.

We then implemented the **handle_inform**, this function handles any inform requests. This function uses a lot of the same things as our previous functions. It allocates space for the new address, it then matches the address with all existing peers on the network, if the address is not already on the network it reallocate space in the network array and inserts the new address. If the address already exist it simply prints this to the client.

The final request type function was the **handle_retreive** function, this function handle any requests to retrieve a file. It starts by opening the requested file and checking if it exist, if the file doesn't exist, it creates a reply header it it sends back to the server informing the server that it was a bad request. If the file does exist however it finds the number of bytes in the file. Based on the number of bytes the file has we make the number of blocks accordingly, we round to the nearest integer, this also insures that at least one block is always created in case the file is very small. The function the uses a for loop to send the file by sending it in smaller blocks, each time creating a reply header with information. When all parts of the file has been sent, the for-loop terminates and we close the file.

Then we wrote **handle_server_request** , this function handles server requests. In this function we extract the request details from the client, turn relevant parts into network byte order and the checks to see which of our three request types have been called. Using if statements we call the correct function with the information from the client.

The last function we had to write is **server_thread** which is the function that runs the server thread. The function opens a listening socket or throws an error if this fails. Once the socket is open, the function uses an infinite for-loop to respond to incoming connections. We have error handling if it can't accept the connection right, otherwise it uses our handle_server_request function to handle the requests. When done with a connection it closes the connection.

5 Tests and shortcomings

To test the functionality of our implementation we started off by testing the basics functionality of our implementation. We did this by opening the first_peer python code with the given command "python3 peer.py config.yaml", after doing this we had a server running and could open our own C client with the command "./peer config.yaml", our client did as we expected and registered to the network as a client as well as requested the two files tiny.txt and hamlet. Now that we knew that our client could work fundamentally as a client we opened a second_peer python code with the "python3 peer.py config.yaml" command again, to see if our implementation could work as a server. The second_peer could register on the server through our C client and could request all the files without issue, we could also request non-existent files without issues.

Now we could test the edge cases of our implementation. We started by creating an empty file, a PDF file, a png picture and a odt file to see if our C implementation could handle sending these files and this worked as it should for all file types. We then opened a new C peer to see if both of the other peers where informed when the new peer joined the network and they were. We then requested files from this new peer and where able to get the requested files from the second_peer.

Throughout our implementation we have attempted to use the network_mutex every time we use the network variable since this variable is a shared resource. We have however not used the retrieving_mutex since we don't use the retrieving_files variable in out implementation and therefor couldn't find a use for it. If we had, had more time we would have liked to try and implement this better so that so our implementation would have been more capable of handling have multiple clients using the file sharing.

And there is the issue with our memory leak, we have some still reachable memory, after the program has closed. The issue is, we have not found a good way to close down the program, so it never reaches the bottom of the main function, where it frees all the allocated memory for, my_adress, network and retrieving_files. This is defintely a shortcoming, since we know the issue, but haven't found a viable solution to his. We could have implemented such that it would react to, ctrl c, and close the server thread, and then close, and free all, if there was more time.

6 Conclusion

We have attempted to implement a client/server for a peer to peer file transferring service. To do this we had focus on multi-threading and socket programming. We also improved existing code and created new functions to improve the clients functionality. The implementation is functional on a basis level, but is still missing some error handling and there are some existing memory leaks that have not been fixed. The assignment tasks 2.1-2.6 have been implemented, but room for improvement and optimisation.