Assignment 4 — Computer Networking (II)

Schmidt, Victor Alexander, `rqc908`
Ibsen, Helga Rykov, `mcv462`
Barchager, Thomas Haulik, `jxg170`

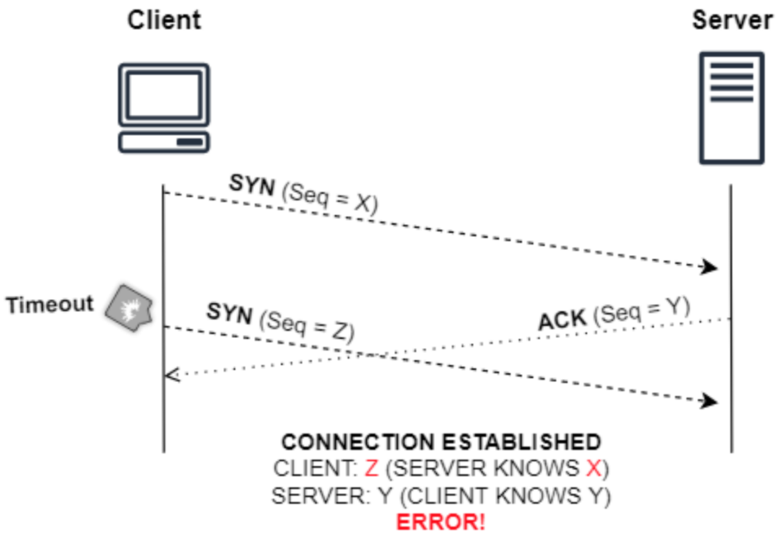
Sunday 16:00, December 4th

1 Theoretical Part

1.1 Transport protocols

1.1.1 TCP reliability and utilization

Part 1: A short answer to that question is yes. The 3-way-handshake in TCP is indeed necessary due to two main reasons: (2) the 2-way handshake can establish communication in one direction only, while one of the core features of the TCP is a full-duplex connection (i.e. both parties can send data to each other simultaneously); and (2) the 2-way handshake cannot guarantee the level of reliability that is ensured by the TCP because the 2-way handshake connection presents potential problems when the acknowledgment message from the server delays too much (cf. Figure 1 from *Two-Way Handshake*):



Figur 1: The two-way handshake process

Part 2: Since TCP connection is a two-way communication between the sender and receiver, both sides must synchronize (SYN) and acknowledge (ACK) each other (cf. Figure 3.39 on p. 254 in the book). And after the connection has been established and data transfer has been launched, the sender needs to send the sequence number (SN) of the packet and the receiver needs to answer with the ACK that confirms receipt of the incoming packet. That is critical to the reliability of data transfer provided by TCP.

1.1.2 Reliability vs overhead

Part 1: In contrast to UDP, the size of the header overhead added to each packet by TCP is considerably larger than that added by UDP. The former ranges from 20B to 60B, while the latter is merely 8B long. That, among other things, makes the TCP connection comparatively slower than UDP, but, in turn, it guarantees reliability.

Part 2: Reliability implies that **all data** is delivered to the receiver. That is, each and every bit sent by the sender is being delivered to the receiver. This is being done via such packet header functions as: (1) *checksums* to detect corrupted packets; (2) *sequence numbers* to detect lost packets; and (3) *acknowledgments* and taking steps to recover the lost data and resend it. In short, reliability encompasses flow control mechanisms, sequencing and sophisticated re-transmission (see below).

1.2 TCP: Principles and practice

1.2.1 TCP headers

Part 1:

1. When a server receives an unexpected TCP packet, it usually responds by sending *a reset packet* back on the same connection. That packet has no payload and with the RST bit set in the TCP header flags. That can happen when, say, the sender sends an initial SYN packet trying to establish a connection to a server port on which no process is listening.

2. One of the TCP reliability guarantees is the delivery of packets in sequential order. That is done via the TCP header fields *Sequence number/SN* and *ACK sequence number*. The sender(can be both a client and a server) sends the *SN* to the receiver that either can correspond to *ISN*(Initial Sequence Number in the handshaking phase), if that is the first packet, or the *SN* that is made of the *ISN* and the first byte of the data/payload. On successful receiving the packet, the receiver replies the sender by setting the *ACK SN* to the next byte it expects to receive. In this respect, the two TCP header fields are interrelated and ensure a sequential processing of the data flow between the sender and receiver.

3. The purpose of the *receive window* (rwnd) is to give the sender an idea of how much free buffer space is available at the receiver. For the sender to be able to send packets to the receiver, the size of the receiver buffer must not be less that the difference between the *LastByteRcvd* and *LastByteRead*. In other words, the receiver will not be able to receive further packets if this number is negative (cf. p.252 in the book). Conversely, it will be able to receive more packets if its rwnd is positive.

4. If rwnd is zero, the receiver buffer is full and the sender is blocked from sending further packets. The problem is that as the receiver empties its buffer, it does not send ACKs with new non-zero rwnd values to the sender, which will halt the data transmission from the sender. To solve that problem, the sender is required to continue sending packets of size 1B when the receiver's window size is zero. These packets will be acknowledged by the receiver. Eventually, the buffer will begin to empty and the acknowledgments will contain a non-zero rwnd value.

Part 2: When the client receives the SYN-ACK packet from the server and initiates data transfer right away, without sending her ACK first, that means that data will flow in one direction, and only the client will be able to send data (the same goes the other way around). The message in the SYN-ACK packet tells the client that the server is listening on port nr and this port is open.

1.2.2 Flow and Congestion Control

Part 1: The modern TCP uses **Dynamic adjustment control**, which is implemented via a network *congestion-avoidance algorithm* that includes various aspects of an additive increase/multiplicative decrease (AIMD) scheme (cf. slides ”Transport Layer: UDP + Reliable Data Transfer + TCP”, slide 46).

Part 2: TCP's strategy for adjusting its transmission rate is to increase its rate (i.e.cwnd/RTT bytes/sec) in response to arriving ACKs — it increases rapidly — until a loss event occurs, at which point, the transmission rate is decreased to the slow start rate. This process is known as the TCP Sawtooth (cf. slides ”Transport Layer: UDP + Reliable Data Transfer + TCP”, slide 52).

Part 3: The congestion window size (cwnd) of the sender depends on the amount of unacknowledged data at the sender. That is, it should not be larger than the minimum of cwnd and rwnd:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$$

Initially, the congestion window is equal to one MSS. TCP sends the first packet into the network and waits for an acknowledgement. If this packet is acknowledged before its timer times out, the sender increases the congestion window by one MSS and sends out two maximum-size packets and so on so forth (cf. slides ”Transport Layer: UDP + Reliable Data Transfer + TCP”, slide 51), until a timeout or the receipt of three duplicate ACKs, which indicates the loss of packets — i.e. congestion on the network. In that case, the threshold is set to one half of the current congestion window and the congestion window is then set to one MSS(cf. Figure 3.7-1: Evolution of TCP's congestion window on *3.7 TCP Congestion Control*).

Part 4: The TCP sender should use the ”fast retransmit” algorithm, known as *NewReno* to detect and repair packet loss (cf. *3.2 Fast Retransmit/Fast Recovery*). This algorithm allows the sender not to reduce its transmission rate to the slow rate. The sender launches fast retransmit in the aftermath of receiving four acknowledgments for an out-of-order packet (one original ACK and then 3 duplicate ACKs) from the receiver, which signal the sender that that packet has been lost.

2 Programming Part

2.1 Protocol

To program our solution, we were given a protocol to follow. We used this protocol as a guideline to create our implementation. Considering the different format of either the client request or client server responses, we had to make sure that all messages had the correct header with the correct length, the correct info and correct structure. We therefore made sure to test that the header and data we receive through a message is correct by checking for e.g for the hash of the data.

In this assignment we had to create a peer, that both acts as a server and a client at the same time. Therefore, concurrency is the only way to go, but this can create problems like race conditions and deadlocks. The way we assure that we do not encounter any of these problems is by locking any shared variables between the threads with a mutex lock. In our case, it was our global variables `network` and `retrieving_files`.

To stress our implementation, you simply need to create a big enough network, with around 9 peers. This will create a segmentation fault, which we believe happens when the network list gets too large. We assumed in our implementation that the network list is small enough to be sent over in a single block. If that is not the case, a segmentation fault will occur.

2.2 Implementation

Client register: We have implemented this by letting the main function create a Client thread. This Client thread will then as first thing send a command to the method **send_message** with the command **COMMAND_REGISTER**. **send_message** will setup a connection with the known peer, construct a header and body containing the client's own IP-address and port and send it all as a request. A reply from the other peer will then be received. The header and body will be extracted from the message, a complete list of peers in the network retrieved and placed in the client's network list.

Client retriever: After the client thread has been registered, it then randomly picks a peer from the network list. Then, it sends a request to that peer via **send_message** with the command **COMMAND_RETREIVE** in order to retrieve a `tiny` text document. As a result, an empty text document will be created, if it did not already exist. After that, the connection is being established, the header and the body (containing the file the want to retrieve) is being constructed and sent together as a request. The peer's reply specifies the following information in its header: the number of blocks to read. In the case of the `tiny` text document, only one block will be read. After retrieving the payload, the client peer hashes it and checks if the data is as expected. Finally, the data can be safely written to the client's own local file after doing the final check of the payload hash.

Server register: To create a server thread, we let the main function call the server thread method. This method will then create a listener on its own IP and port and keep waiting in a infinite loop for receiving any server request. On receiving the server request, a server thread is being created. It extracts the header and the body and then determines the requested command — in casu, a register command is received and **handle_register** is called with the IP-address and the port of the calling peer. The method starts by checking whether the IP-address and the port are valid and whether they already in the network list or not. If the peer is not already in the network list, the method will update the network list with that new peer. The next step: the header and the body (including the list of all participant peers in the network) are created and sent back to the calling peer. Afterwards, the method will inform other peers in the network, by sending a message with a full list of peers in the network via the command **COMMAND_INFORM**.

Server inform: When the server thread receives the `inform command`, **handle_inform** is called with the new peer's IP and port number. **handle_inform** retrieves the IP and the port and checks if the peer is valid. It also checks, if that peer already exists in the network list. If the peer is valid, but is not on the network list, **handle_inform** updates the list with the new peer.

Server retrieve, large and small files: After the server thread receives a retrieve command, **handle_retrieve** is being called with the requested file name. **handle_retrieve** calculates the number of blocks to split the file into and creates each block with its own header (incl. the total number of blocks, the length of each block and the serial number of the given block). The body contains data from the file. `tiny.txt` requires one block only. `hamlet.txt` requires many blocks because the length of the file is larger than the allowed length of a message. Each sent block is accompanied by a message.

2.3 Tests

What was tested?

As mentioned in the assignment, we have made the following test-cases:

1. Can we connect to a given peer? (expected: yes)
2. Can we connect to a given peer with the same IP and port twice? (expected: no)
3. Can receive info about other peers on the network? (expected: yes)
4. Can we request and receive a small file? (expected: yes)
5. Can we request and receive a big file? (expected: yes)
6. Can we request and receive a file that does not exist? (expected: no)
7. Can we register new joining peers? (expected: yes)
8. Can we register joining peers twice with the same IP and port? (expected: no)

9. Can we inform the rest of the network about new joining peers? (expected: yes)
10. Can we send a small file to a peer? (expected: yes)
11. Can we send a big file to a peer? (expected: yes)
12. Can we send a file that does not exist to a peer? (expected: no)
13. Can create a network with 5 peers? (expected: yes)
14. Do all 5 peers receive info about new joining peers? (expected: yes)
15. Can we connect to DIKU's peer? (expected: yes)

The tasks listed up above have been tested on MacOS imperatively in the `main` method. It is important to note, that the "expected" outcome is defined by ourselves and means the result we believe to receive on having tested those tasks.

All of our tests passed as expected, except the last one # 15, where the code should have been tested on a real-life peer, the DIKU server. The reason behind that could be the fact that our firewall does not allow the connection with an external network. We attempted to connect to DIKU's server both from home and on the campus.

The response: `(Server response) Open_clientfd error: Connection refused`

To conclude: we are confident in our implementation as all tests have gone through as expected. We use `mutex locks` for the shared variables, ensuring no deadlocks or race conditions.

2.4 Shortcomings

2.4.1 Technical shortcomings

Not able to connect to DIKU's server: As stated in our test section, we were not able to connect to the DIKU server, which is a shotcoming. We not sure what is exactly is stopping us for connecting, but we did try from home and at DIKU.

2.4.2 Reflections

We do not have many shortcomings this time around, all our tests except one passed, we have followed the protocol as described. Perhaps the code itself can be hard to read at times, but the assignment is also partly to blame for this, as networking and threading often dilutes the code with locks / connections / protocols, etc..