Compsys - A4

Christian R (srw812), Nicolai N (dwx635) og Andreas J (cmn945)

December 5, 2021

Theoretical

Transport Protocols

TCP Reliability and Utilization

Part 1
The three-way handshake is necessary, because having a two-way handshake would mean that either the client or the server can send data but not both.
Part 2
TCP ensures full-duplex by ensuring that data can be send from process A to process B and vice versa at the same time if there is a connection between the two hosts.

Reliability vs Overhead

Part 1
UDP does not add overhead, since data is continuously sent to the recipient, whereas TCP uses a larger overhead ensuring a more reliable connection.
Part 2
There is a guaranteed delivery of data to destination router and even retransmission of lost packets are possible at the cost of being slower than UDP

TCP: Principles and Practice

TCP Headers

Part 1
1.1
The RST bit can terminate a connection. This might be due to a fatal error such as if one host has closed the connection but another host wish to continue with the data transfer.
1.2
The sequence number refers to how much data has been transferred since it is included in each packet sent by the client. The acknowledgement number indicates whether the data transfer was successful or not by sending it back to the client.
1.3
The purpose of the window is to tell the host on the other side of the connection how much data can be transferred before an acknowledgement has been received. A positive window size signifies that there is still room for data to be sent.
1.4

A window size of 0 tells the sender that the receiver cannot receive more data before the the window size has been increased. The sender will find out since the receiver is halting the data transfer.

Part 2

?

Flow and Congestion Control

Part 1

TCP does not use network assisted congestion control. Instead, TCP employs a slow down policy in case that the network cannot follow the sender in terms of speed. If the network cannot follow it will tell the sender to slow down to avoid congestion.

Part 2

TCP determines the transmission rate when doing congestion control by implementing a congestion window that keep track of the transmission rate by this formula

$$R = \frac{w}{T}$$

where R is the transmission rate, w is the congestion window, and T is the round-trip latency

Part 3

The congestion window changes the transmission rate depending on the traffic. If the connection is stable, the congestion window will increase and thus increasing the transmission rate. If it overflows the congestion window will decrease and thus decreasing the transmission rate. All this is controlled by the congestion control - algorithm. The transmission window is initiated with a slow start to avoid transferring too much data such that the network cannot transmit all of it.

Part 4

After receiving three duplicate ACKs for the same segment, we assume that the packet is lost and will retransmit immediately without the acknowledgment from the opposing host, thus allowing for a fast retransmit.

Report

Introduction

In this assignment we are to implement a fully functional peer in c to be be a part of a peer-to-peer network using .cascade files to share and download files through a tracker. The peer should be able to work as both a server and a client simultaneously for different .cascade files.

Technical Implementation

The first goal of the implementation is to subscribe our .cascade hashes to the tracker. We do this through a subscribe method which is really just a copy paste of get_peer_list but with a 2 in the header instead of a 1 marking it as a subscription, while this is going on we are also parsing the files, which we are marked as having the .cascade files for.
Please note that already at the code mentioned just above, the code breaks and we simply cannot tell what is causing it to break, so the rest of this report will speak in purely theoretical terms about what our intentions were, rather than what has actually happened
This is put into a struct which we pass to out 2 threads which now split, An upload thread and a download thread, a design choice, which we will discuss in more detail below. These 2 threads then have each their own worker threads, both containing infinite loops.
The download thread will run an infinite loop where it continually loops through the files which it have marked as looking for then trying to find them from a peer and downloading them. Note here that the peer does not know which blocks/files it already has completed until it checks and so it will keep trying to look for things to download even if it has gotten all it was looking for
The upload thread gets a bit more advanced than the download as this is the new functionality we are developing. First we simply listen on the ip and port until we receive an attempt at a connection. We then initialize a buffer to communicate and pull the request, putting it through a quick search to ensure it is on the right form. We can then pull the actual usual information from the request, like the block number and the hash for the file.
With this information we then search our files to find the matching hash and pull the block. Here we remember to do sensible error checking, like if the block number is bigger than the block count for our file and if we even have the hash which the peer is trying to get from us.
If everything then checks out we format our response header appropriately and tag along the block at the end of it, then freeing the allocated memory and closing the sockets we opened

Non-blocking

The main way we attempted to make the peer non-blocking was by simply providing complete separation of downloading and uploading through having 2 different threads handle each, that way it can run each in their own time and never have to worry about one locking the other one out, especially since, while they share some data, they never modify that shared data, and therefore we avoid any nasty race conditions without having to use any mutex or the like, further lowering the chance that the peer might block on doing either of its required 2 tasks.

Design testing

the code was not able to be formally tested as it did not in the end run properly. However for what we would have done, we'd first spin up a tracker and 2 peers with a mix of files and then download them with our own implemented peer, then kill the 2 other peers and spin up a new peer to download from our peer to test the uploading

Conclusion

In the end our implementation did not properly work, however we personally felt quite good about the work we did and think it must be some small weird thing which is causing it to segmentation fault, but we just can't find it ourselves. That being said we think we are reasonably happy with how the project turned out in the end, even if there are some extra hidden bugs, which we can't even debug because the program breaks so early from the previously mentioned problem.