Assignment Report

OVERVIEW

This report will aim to outline the general design principles chosen for the P2P Network that was simulated, the reasoning for the choices, and the drawbacks of such choices, and other constraining factors. Furthermore, any deviations from the specifications or shortcomings will be explained. A breakdown based on the 5 major steps outlined in the specifications will allow for a discussion of any issues specific to a step, which are independent of the programming choices and issues. The report will finally be broken down by the files used to logically separate the code required to implement the desired functionality. These sections will be further divided into TCP and UDP sections, if necessary.

The link to the demonstration is `https://youtu.be/LpPBzpToPdE`.

OVERALL CHANGES

There are a few overall design changes I would make, had I had the time and ability to get it working correctly for the implementation. The first and foremost major change I would implement would be the use of shared memory to implement locks for sychronisation of certain activities, rather than relying solely on the `sleep` function. At a high level, this is bad practice, as it is not actually a guarantee that it will synchronise the two processes/threads into the right order, as without control over the underlying OS Scheduler, the sleep cannot guarantee it will order the processes/threads correctly, as one may get no CPU time, whilst the other is sleeping, and hence can end up still in the incorrect order. Later in the report, I will give better examples of where this occurs and how I would change it.

Another change I would make would be to have code to handle errors such as connection failure, and any other large scale errors, so rather than just printing the error and exiting with an error code, have some recovery code, to determine if the connection failed due to a peer quitting, and if so then updating successors and performing the necessary functions. Or on the other hand, working out if it is an issue with the peer attempting to connect, and have some error recovery code. However, there were simply too many edge cases and issues, that simply exiting with an error message and error code, was the cleanest and most time effective solution in this case.

The other major changes I would implement would be to allow the servers, both UDP and TCP servers, to have the ability to spawn a thread for every incoming request, and let that thread handle and perform the computation. This obviously frees up the server to handle more requests, and prevents the server getting stuck on only doing a file transfer for one peer, or something to that effect. Again, specifics will be mentioned later. Another overall idea would be changing from using C to use Java instead. For socket programming, and string handling, I have found C to be far more complicated and difficult, as well as less robust. Java provides a much higher level of abstraction than C, and is far simpler to handle the messages sent as strings.

SPECIFICATION STEPS

Step 1: Initialisation

No issues or design sacrifices had to be made with this step. It was fairly straightforward, and doesn't warrant any considerations about changes I would implement if I had the time and/or ability.

Step 2: Ping Successors

This step was again fairly straightforward in terms of implementation. One issue did arise when transferring the code to the CSE machines. For some reason, still unbeknownst to me, the `strtok` function I beleive was somehow manipulating different memory addresses to what it was supposed to. This is just a guess, as depsite extensive testing, I could not figure out what was happening, and so simply called `strdup` on the string that was being effected to further ensure that `strtok` was operating on a copy of the string. This problem was non-existent on my machine at home. However, the `strdup` solution worked fine and resolved it. In terms of ping parameters, the implementation sends out 10 messages, and declares the successor dead if it doesn't respond to any messages. This aims to give a high degree of certainty that the peer is in fact dead, and the network is not dropping packets. If the peer responds to any of the messages, the peer is declared alive, and no further pings are sent. This choice was made to free up the network for file transfer and other tasks. The timeout value is set at 1 second. Successors are pinged every 60 seconds. Had I been able to figure out how to correctly implement a server that spawned thread to handle every incoming request, I would have constant pings in the background, to give a more realistic view of the real time changes of the P2P Network.

Step 3: File Transfer

This was probably the toughest step to implement, and required a lot of fiddling around to get correct. Firstly, in my implementation, I made the choice to handle when the requesting and responding peers were actually the same, and made the decision to not transfer the file in this case, as the peer already has the file. Furthermore, the way in which I determined who held the file was based upon a peer calculating if their immediate successor held the file, and if so, notifying them in the TCP forward message. This seemed to be the most logically consistent method, when considering the circular linking, especially when it came to dealing with the region between the smallest and biggest peers. In terms of the stop and wait protocol, this was implemented smoothly, with a timeout for receiving ACKs in the sender of 1 second again.

A few choices were made in this step that were required for the implementation. Firstly, the file was transferred using server ports of 30000+i, where i is the number of the peer. This was to prevent interfering with the Ping messages being sent also over UDP. The next choice made was to start the sequence number from 0, not 1 as the example log shows. This was chosen because the first expected byte is at position 0, not position 1. Also, when the sender receives an ACK, the number of bytes is set to 0 in my logs, not MSS, as the ACK contains no bytes of data. The same applies in the receiver's logs, when they log a snd action, they are not sending any bytes. Finally, on the logs, the formatting using the tab special character works nicely for `cat`, but not so much for gedit and other file viewers. When transferring the bytes of the PDF file, I included a header of 20 bytes. The header included the sequence or ack number, depending on who sent the message, and a bit to indicate if it was the final segment or not, so the receiver can close the file correctly. The last thing I would have changed for this section would be the addition of thread handlers in the server, to allow multiple file transfers at the same time, and not tying

the server up to handle only one at a time. This would be true not just for the file transfer section, but for both the TCP and UDP server, if I had the time to make the implementation of threads to handle incoming requests correct.

Step 4: Peer Departure

This step was implemented quite smoothly in the end, after reworking how I updated predecessors. Initially, I had some complex functions to determine how to set a peer's predecessors based on which peer pings them. As a result, on peer departure, I initially had to send messages to the departing peer's successors so they could make that predecessor as dead, and wait to get pinged to update it. However, once I reworked it such that I sent whether a peer was pinging its first or second successor, it became very easy to update the correct predecessor. This removed the need to also send to a departing peer's successors.

Step 5: Kill a Peer

This step was quite smooth, without any real design choices or issues. As mentioned earlier in Step 2, the peer is declared dead if it doesn't respond to any of the 10 messages. The one main issue with this section occurs in my implementation if a peer is noticed to be dead when the peers are pinging to update their successors from a previously identified dead peer. This identification will have to wait another 60 seconds before it can resolve any dead peers that occur in the successor ping stage of a previously dead peer's resolution. Ideally, if a peer was noticed to have died during the one-off ping to resolve a prior peer death, this would trigger another peer death resolution, and so on until no more peers were dying. However, this was not chosen to be the case, as it could end up spiralling some specific peers into a constant state of resolution and pinging, if the right timing sequence of peers dying was entered. My current implementation ensures that only one round of dead peer resolution occurs every 60 seconds, to ensure that in the time between, the peers are free to engage in file transfer.

The resolution process required the peer whos's second successor was the dead peer, to wait for the peer who's immediate successor died. This is because otherwise, incorrect information would be passed back to the peer who's second successor was declared dead. Thus, a sleep was added to allow the peer who's first successor had died to resolve their successors, and then pass the information to the peer who's second successor had died. Ideally, this would be done through semaphores and sychronisation, not sleeps.

FILES

strhandle.c

I would have preferred to error check this file and its functions far better, but C is quite difficult to do so, and thus this gives good weight to the argument of changing it all to Java.