COMP9331 Assignment

Ilia Chibaev z3218424

Program implementation:

My Java implementation of the assignment consists of 8 classes. *cdht* is simply used as a main method which creates a *DHTPeer* object from the input arguments.

Each instance is a *DHTPeer*, which maintains 2 successor integer port variables and 2 integer predecessor port variables. The successor ports are stored in separate variables to maintain the order, while the predecessor ports a written to an integer array without any consideration of their order. It also has a *Server, Client,* and an *InputListener* object.

The *Client* object has a runnable *PingSender* object. The *Client* runs the *PingSender* at a set time interval of 10 seconds on a *ScheduledExecutorService*. It can also send data over TCP with the *sendData* method, and to send a request and return a response with the *sendRequest* method. The TCP response timeout for the *Client sendRequest* method is 2 seconds.

The *PingSender* object pings both *DHTPeer* successors and increments the corresponding sequence number if it receives a response. The UDP socket for the *PingSender* is on port 50256 + the peer id. This is so it does not conflict with its *UDPListener* socket which is 50000 + the peer id in accordance with the assignment specifications. The UDP socket times out after 2 seconds of waiting for a response to the ping message. If it has not received a response in 3 pings (determined by the difference between the sequence number of the last ping request sent and the last ping reply received), it assumes that the successor is dead and messages its other successor to get information on how it should update its successors (see Kill a peer section). Once it has updated its successors it resets the sequence numbers for the first or second peer (depending on which one dropped) to 0.

The *Server* object has a *UDPListener*, and a *TCPListener* object. The *UDPListener* object runs in a thread and is constantly listening for incoming UDP connections. When it receives a message, if the at least one of the *DHTPeer* predecessor ports is still not populated, it will add the port to the list. If both ports are recorded, but it receives a message from a new peer, it flushes its memory of its predecessors and starts populating them again. Once it has received a message, it will respond with its own peer id and the sequence number it received with the ping message.

The *TCPListener* objectruns in a thread and is constantly listening for incoming TCP connections. It understands 4 commands: “request“, “File“, “quit“, and “successor“. “request” and “File” correspondingly are for receiving a file request and receiving a file in response to a request. “quit” lets the *DHTPeer* know that a successor is leaving gracefully. “successor” is asking the peer what it’s first or second successor id is. The TCP timeout for the response to a “successor” message is 2 seconds.

Finally, the *UserInputListener* object runs in a thread and is constantly printing a new line to accommodate user input and scanning for input. It understands 2 commands: “request” and “quit”. The “request” command initiates the *DHTPeer requestFile* method with the input file name at the end of the input message. The “quit” command initiates the *DHTPeer quit* method.

*Ping Successors:*

A peer begins to ping its successors 2 seconds after initialisation. The interval between pings was set to 10 seconds to maintain a balance between allowing the user enough time to enter a command (though the command will still be valid if a ping message interrupts it half-way through typing) and sending ping messages often enough to detect whether a successor has dropped in a reasonable amount of time.

The message formats for the ping messages are as follows:

* Ping message: 3 character id of the ping sender, then a 2 character sequence number
* Ping response: 3 character id of the ping responder, then a 2 character sequence number

*Requesting a file:*

The message formats relating to request a file are as follows:

* File request message: “request”, then a 4 character file name, then a 3 character id of the requesting peer, then a single character “0” or “1” representing whether the peer’s first successor is expected to have the requested file
* File request forward message: contains all the same information as the file request message, but with the last character updated with whether this subsequent peer’s first successor is expected to have the requested file
* File request response message: “File”. then the 4 character name of the file requested, then a 3 character id of the sender peer

To request a file, a peer sends a file request message using TCP to its first successor. If this successor has the file then it will establish another TCP connection and send a file request response message to the requesting peer. Otherwise, if the successor does not have the file, it forwards the request message to its first successor.

The assignment specification states that requests should be forwarded only to the first successor. I initially implemented the ability to forward the request to the second successor where appropriate but commented out the lines in *DHTPeer.checkFileInSuccessor* which allow for this in order to conform with the marking guidelines.

*Peer departure:*

When a peer quits, it sends a message to its predecessors using TCP in the following format: “quit”, then the 3 character id of the quitting peer, then the 3 character id of its first successor, then the 3 character id of its second successor.

When a peer receives a "quit" message from one of its successors, it calls its *updateSuccessors* method. If the message is from its first successor then it moves its second successor in the place of its first successor and sets its second successor to the quitting peer's successor. If the message is from its second successor, it simply replaces its second successor with the first successor of the quitting peer.

*Kill a peer:*

In my implementation, a successor must not respond to 3 ping messages before the peer decides that it has dropped, so the time it takes a peer to discover a dropped successor is 30 seconds. I chose 3 messages to be missed before deciding a successor has dropped because it allows for some reasonable amount of packet loss on a real network connection and limits the time to discovery of a dropped peer to 30 seconds.

In order reconfigure its first or second successor, the peer will send a “successor” message using TCP in the following format: “successor”, then a “1” for the first successor or “2” for the second successor. The receiving peer will simply respond with the id of the its requested successor.

When a peer’s first successor has dropped, it will set move its second successor into the place of the first successor, and message this remaining successor what is its first successor and place this peer id as its second successor.

It is a little more complex what happens when a peer’s second successor has dropped. Because it doesn’t know whether its first successor has realised that its first successor has dropped yet and updates its own successors. For this reason, it sends a message to its first successor asking what is its first successor. If its first successor is the same as the requesting peer’s second successor, then it has not yet been updated and the requesting peer sets its second peer to its first successors second peer. Otherwise, the first successor has been updated, so the requesting peer sets its second peer to its first successors first peer.

YouTube demo:

I am running the demo on the CSE machine weber via SSH. You can clearly see in the video that I SSH into weber then launch xterm in an VcXsrv X server. I have asked whether I could do this on the course forum and Ali said that this was fine. Below is the link to the YouTube demo:

<https://youtu.be/Y5EwKH8SKAk>