COMP3331 Assignment Report
Bethia Sun z5165555
Programming language and version used: *Python3*
Link to demo: https://youtu.be/SLn2NF--9Dc (with voiceover)

Section 1: High-level Overview of System Design

For this assignment, I implemented the P2P network and its functionality by designing one class with multiple attributes – the Peer class, and writing multiple functions (some of which are threaded) within that class that could be invoked in order to deliver the required functionality of the system.

Section 2: Brief Description of How System Works and Implementation Details **with some details omitted due to word limit

Step 2: Ping Successors:

I created two threaded functions inside my Peer class to implement the key ping functionality required by the system. One of these threaded functions was a UDPListen threaded function that is immediately invoked by main when the program starts. The UDPListen thread listens indefinitely on the UDP port each Peer is bound to to (50000 + peer ID). It listens for two types of messages – ping request messages and ping response messages, identifying the type of message by their headers (this is further discussed in Section 3: Design choices). If a Peer receives a ping request message, I invoke the sendPingResponse function which sends a ping response message over UDP (as seen in line 136) to the appropriate requesting Peer. Every time a request message is received, I also invoke an updatePredecessors function, which ensures that the two immediate predecessors of each Peer are dynamically determined based on the ping request messages received.

The second of the two threaded functions I designed to implement the ping functionality of the system was a sendPingRequest function. This threaded function is also invoked by main as soon as the program starts. This function continually sends ping request messages from a Peer to its two immediate successors over UDP (as seen in lines 146-151).

In this way, a Peer is able to both send Ping request messages and listen to any ping request/response messages it receives simultaneously.

Step 3: File Transfer:

For the file transfer component of the assignment, I implemented the key functionality required in the following way:

For the handling of the input, I created a threaded function within the Peer class called handleUserInput, which indefinitely waits for user input to be typed into a Peer's own terminal. If the user input contains the keyword "request", then a function call to findFileLocation is invoked, which sends the file request message to the Peer's immediate successor over TCP by using the Peer's TCP client socket object.

Simultaneously, in each Peer, the threaded TCPListen function I created is listening on the TCP socket object that is bound at the port 50000 + Peer ID. The threaded TCPListen function manages messages sent to the Peer over TCP by identifying the type of message that is received and invoking appropriate functions for that message. If the type of message received is a file request message, then, the findFileLocation function is invoked, which either forwards the file request message over TCP to a Peer's immediate successor (as can be seen in lines 319-324), or sends a file response message over TCP directly to the requesting Peer if the current Peer is the closest Peer to the file location (as can be seen in lines 376-382). So, in this way, the file request message is able to be forwarded over TCP from each Peer to its immediate successor until the message reaches the Peer closest to the file destination.

Once the message reaches the Peer closest to the file destination, the Peer closest to the file sends a file response message, which notifies the requesting Peer that it will begin sending the file data. The responding Peer then invokes a sendFile function, which sends the file to the requesting Peer over UDP (as can be seen in line 433) to the requesting Peer's fileReceiverSock object (a UDP socket object bound to the port ID + 60000). The sendFile function encapsulates MSS bytes of file data with a header containing the ISN (set to 1), the packet's sequence number, the MSS and the keyword "START", which signifies the end of the header and the start of the file data. The sendFile function ensures that the data transfer over UDP is reliable by retransmitting the appropriate packet if a timeout has occurred due to a packet being dropped. It also implements stop and wait behaviour by ensuring that no packets are sent unless the corresponding acknowledgement has been received.

On the other side of things, once the requesting Peer identifies that another Peer is sending it a file response message in its threaded TCPListen function, it invokes a receiveFile function. The receiveFile function extracts the header from the message so that the requesting Peer can send the appropriate ACK (which is determined by the ISN, the sequence number and the MSS) as well as incrementally decoding the message until it reaches the word "START". The requesting Peer then writes MSS bytes following the word "START" to a file with a filename in the format "received_file_[fileID]_[requesting Peer ID].pdf" (the log files are in a similar format). It does this until it receives a message from the responding Peer that the file has been sent, at which point the file is closed.

Step 4: Quit (Graceful Departure):

I have handled graceful departure by using the handleUserInput threaded function (as described above in Step 3). If the keyword "quit" is in the user input, the function, sendPeerChurnMsg is invoked to notify the Peer's immediate predecessors that the peer will be departing the network, as well as invoking a function to terminate any running threads.

The sendPeerChurnMsg creates a TCP socket object (as seen in lines 376-380) to send a departure message to the Peer's two immediate predecessors. The departure message contains the number of updates to the Peer's successors that the immediate predecessor must make, as well as what these successors will be.

On the predecessor side of things, the peer churn message is picked up by the threaded TCPListen function which listens to any incoming messages on the Peer's TCP port. This prompts the Peer to inspect the message contents to determine how many successors it needs to update and the identity of these successors. The Peer will then reinitialise its successor peers so that its self._immediateSuccessors attribute contains the identities and addresses of the appropriate successors. In this way, when the Ping sends ping request messages to the successors in its self._immediateSuccessors attribute, it will be sending ping request messages to the correct Peers, maintaining the structure of the circular DHT.

Step 5: Kill a Peer (Ungraceful departure):

To implement kill a Peer, I ensured that in any of the threads if a Keyboard Interrupt occurs that all the threads will terminate and all the sockets bound to a specific port will close. This prevents the 'killed' Peer from communicating with any of the other Peers.

For the predecessors of the killed Peer to detect that the Peer has been killed, I implement logic in UDPListen to determine the number of ACKs the two immediate successors of a Peer have missed. If the missed ACKs of any successor of the Peer is equal to or exceeds 7, this indicates to the Peer that the corresponding successor is dead.

The Peer with a killed successor then makes any necessary updates to its first successor and then sends a FindSuccessorMsg to its new first successor over TCP using a TCP socket object (as can be seen in lines 245-253). This message asks for the Peer's first successor to supply the Peer with its own first successor, which the Peer can then use to update its second successor. Both the message requesting for a Peer's successor and supplying a Peer's successor are sent over TCP so the necessary actions once these messages are received are invoked in the TCPListen thread.

Again, once the Peer's two immediate successors are properly updated, the structure of the circular DHT is maintained as ping messages are only being sent to the correct successors.

Section 3: Design Choices, Including Message Design

A design choice I made that was key to my program's functionality was to create two threaded functions TCPListen and UDPListen, which would listen to all incoming messages of a TCP nature and UDP nature respectively, and then, depending on the type of the message, invoke certain functions to perform certain results. This design choice was motivated by the simplicity of such a system, though if time permitted, I would have liked to create more classes. As mentioned in Section 1, I decided to design my system so that there would be one Peer class for simplicity, though this inevitably increases the number of dependencies between functions.

In terms of message design, I designed all messages to have all have ASCII decodable identifiers in the first 10 bytes of the entire message (e.g. “PING”/”FILE”/”HEADER”/”ACK”/”DEPARTURE”) so that my identifyMsg function (lines 88-131) could be invoked to identify what type of message was being received. In this way, my threaded UDPListen and TCPListen functions were able to identify the type of message being received and call the appropriate functions to ensure the appropriate actions were taking place.

A particularly important example of message design in my program was designing the messages encapsulating the file data. I started any messages containing file data with the keyword “HEADER”, to importantly alert the Peer receiving this message that only the first few bytes of the message could be decoded (as the pdf data is not able to be decoded into ASCII). To overcome the fact that messages containing file data were not able to be fully decoded into ASCII, I ensured that in my receiveFile function, the requesting Peer only decodes the header and not the file data by incrementally decoding bytes until the keyword “START” is reached. The keyword indicates all remaining bytes in the message would be pdf data, so appending this keyword before the start of the pdf data ensured that there would be no attempts to decode the pdf data into ASCII and that each Peer was able to write all pdf data to a file properly (as it knew that all MSS bytes after the “START” was pdf data).

Less significant design choices I made included: sending pings every 7 seconds (to not overwhelm the evaluator) as well as declaring a Peer to be dead if it has exited the circular DHT ungracefully if 7 or more ACKs have been missed.

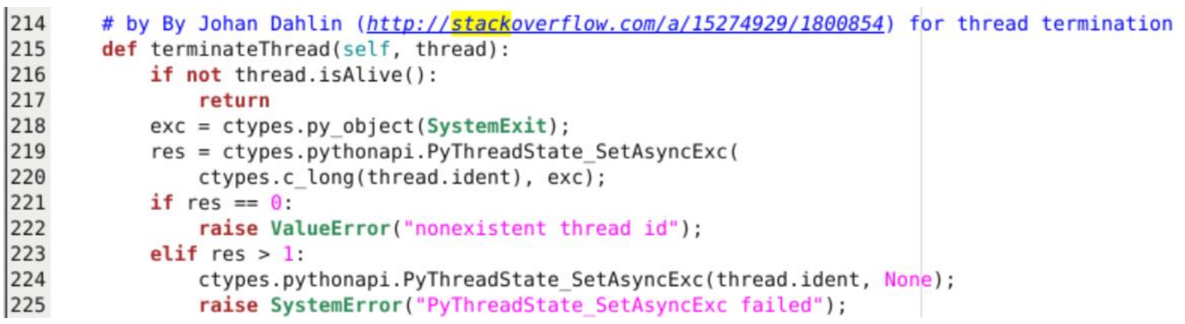
Section 4: Possible Improvements and Extensions to Program, Potential Errors

Possible improvements and extensions to program: The style of the code I have written could definitely use some improvements as there is only one class and many of the functions within that class have tightly interwoven dependencies. If time permitted, I would have liked to create more classes to reduce the dependencies between functions. Furthermore, I have several code redundancies/inefficiencies due to creating multiple dynamically bound socket objects– for example, to send the peer churn message, I create a TCP socket object, and to send the find successor message, I also create another TCP socket object, which is a bit redundant. An improvement would be perhaps creating more threaded functions to send messages from the same TCP/UDP port simultaneously to avoid this.

Circumstances under which the code does not work: Sometimes (very rarely) I get a Unicode decode error from the codecs module I have imported and I am not too sure why, since the first 10 bytes of all the messages I am sending is ASCII-encoded and I am only decoding the first ten bytes of every message unless it is a fully decodable message. I have tried debugging but I have not been able to identify the cause. I do not think this occurs on the CSE machines. I may have fixed this however, as it has not happened recently. However, if it happens during the demo, please could the marker run my program again (or wait a few moments before rerunning the program), as this generally fixes the error. Additionally, very rarely I get a socket in use error despite closing all sockets and specifying the reuse address option, though this hasn’t occurred on the CSE machines (only on VLAB). Please could the evaluator run my program again if this occurs/wait a few moments as this generally fixes the problem.

Section 5: Borrowed Code (Indicated with Source Reference)

I borrowed code from StackOverflow below (reference link: http://stackoverflow.com/a/15274929/1800854) on how to terminate threads in lines 214-225 of my program:



```
214     # by By Johan Dahlin (http://stackoverflow.com/a/15274929/1800854) for thread termination
215     def terminateThread(self, thread):
216         if not thread.isAlive():
217             return
218         exc = ctypes.py_object(SystemExit);
219         res = ctypes.pythonapi.PyThreadState_SetAsyncExc(
220             ctypes.c_long(thread.ident), exc);
221         if res == 0:
222             raise ValueError("nonexistent thread id");
223         elif res > 1:
224             ctypes.pythonapi.PyThreadState_SetAsyncExc(thread.ident, None);
225             raise SystemError("PyThreadState_SetAsyncExc failed");
```