CS3331 Assignment Report

By Katrina (z5211336)

Video Link: `https://youtu.be/7Hh0qGG7Aq8`

Testing Environment

This program is written in Python3 and is tested with Python 3.7.2.

Program Design

The program is running with **four** threads, and each of them is infinite loop. They are: **FileHandler()**, **pingSender()**, **pingListener()** and **TCPHandler()**. Except for those, in the program, there is also an infinite loop that **monitor the input from terminal**.

All packets, segments, messages are **class instance**, and will be dumped and loaded by **pickle** during sending and receiving process.

FileHandler

Handle file transferring. Only start when a file request/response are successfully received, and finish when the file is successfully sent. It also check FINISH flag during transmission. FileHandler() is transferring file using Stop and Go. **Each ACK = len(recvContent) + recvSeqNumber.** If there is a packet loss happened in sender side, the receiver won't sending corresponding ACK, then timeout will occur at sender side, and sender re-transferring the packet. Sender will only send next packet when the ACK of last packet is received, and **newSeqNumber = recvAckNumber**.

The function behaves differently by identifing if the flag `IM_SENDER` or `IM_RECEIVER` (both Boolean) is `True`. Hence, **this program allows a peer to be both file sender and receiver**. As assignment specification does not specify a particular port for file transmission, to avoid concurrency, file packets are received from port `60256+myId` (it comes from the maximum number of peers `255`) and sent from random ports.

Also, both sender and receiver have their own global variables. Sender has `lastSentSeq` and `lastRecvAck` to record the last sent packet's sequence number and the last received acknowledgement number. Receiver has `lastRecvSeq` and `lastSentAck` to keep track of the latest packet that received and expected next packet.

Both sender and receiver have ability to **ignore duplicate packet** by comparing the incoming seq/ack number and recorded last received seq/ack. They are also responsible for the **log file**.

`FileSegment` instance include:

Attributes	Description
seq	Sequence number. ACK and FINACK packets don't have sequence number (None).
ack	Acknowledgement number. PAYLOAD and FIN packets have no ack number (None).
flag	Flag to indicate type of packet (ACK, PAYLOAD, FIN, FINACK).
content	Payload of file segments. ACK and FINACK packets don't have content (None).
mss	Maximum segment size, which is read from command line.

pingSender

The thread will consecutively send ping to its successor1 and successor2 every 20 seconds. Each time sending a ping message, it'll choose a random port and apply drop rate. It also uses **sequence number** on ping messages which allows its successors to keep track of its activity.

`PingMsg` instance include:

Attributes	Description
seq	Sequence number of the ping message.
id	Sender's id.
flag	Can be 'REQUEST' or 'RESPONSE'.
myPos	Sender's position ('Prev1'/'Prev2'/'Succ1'/'Succ2').

pingListener

A thread that keep listen to port `50000+myId` for ping message. It will immediately update its predecessors every time receive `REQUEST` flag ping message. And also keep track of latest received sequence number from successors every time receive `RESPONSE` flag.

If it finds that one of its successors is **ahead of another one by 4 sequence numbers**, the one having lower sequence number will be regarded as being killed. If the dead peer is its Succ1, it'll send information request to its Succ2 immediately and update successors; if the dead peer is its Succ2, it will first send information request to its Succ1. However, **if the dead peer is still in Succ1's successors, it won't receive and update successors information**. It will then *balance* the sequence number record by incrementing Succ2's sequence number record by 2 (i.e. wait 2 more pings for Succ1 to update its successors).

Once it successfully updates two successors, the last received sequence number record will be overwritten by new successors' ping response.

TCPHandler

This thread will continuously listen at port `50000+myId` for connection. When there's a message come in, it'll firstly identify the flag then behave as follow:

Message Flag	Behaviours
FILE_REQUEST	Read the file name and check if the file is supposed to be stored here. If yes, check if the file exists. It will print propriate Error message if the file does not exist. Then send `FILE_RESPONSE` directly to requesting peer and start `fileHandler` thread.
FILE_RESPONSE	Once the requesting peer receive `FILE_RESPONSE`, it will also start the `fileHandler` thread.
QUIT_NOTIFY	Receiving this means the sender is quitting p2p network. Update its successors according to the provided messages in the packet.
INFO_REQUEST	When a peer is requesting for its successors information it'll receive this. As the dead peer is notified in message, the peer check if its successors contain the dead peer. If the dead peer is in successors, it won't send back wrong information; else, it responds with its successors' information and `INFO_RESPONSE` flag.
INFO_RESPONSE	Once the peer gets this, it will update its successors immediately.

`TCPMsg` instance include:

Attribute	Description
id	Sender's id.
flag	Message type as defined above.
myPos	Sender's position ('Prev1'/'Prev2'/'Succ1'/'Succ2').
msg	Various information needed.

Others

In main program, there also an input monitor which is an infinite loop to extract user input. It will validate user input and print appropriate messages if input is invalid. If the messages is "quit", it will set the global variable `FINISH` to `True` to notify all threads to safely exit and close sockets. It'll also capture `Ctrl+C` as KeyboardInterrupt and perform safe quit as above.

Constraints

1. As the xterms are opened subsequently, there is delay for some peers joining the network, so it might take some time to update its predecessors. During this time, "request file" process is not working as the "checkFile" function will fail to identify if it's the expected peer.
2. The gracefully quit process is slow because it's done by setting global variable `FINISH` and is checked by each infinite loop. However, it's thread safe and guarantee to close all opened socket before terminating the program.