# COMP3331: Computer Networks and Applications

## ASSIGNMENT FOR TERM 1, 2024

Name: Phoebe Tandjiria
zID: z5444555

Language: Java
Files submitted: Sender.java, Receiver.java, report.pdf

## PROGRAM DESCRIPTION

The purpose of this project is to implement Simple Transport Protocol (STP) over the UDP protocol, ensuring that the delivery of data between a pair of hosts is reliable in the case of packet loss.
Two programs are used in this project:

- **Sender.java –** The sender initiates connection set-up/connection teardown phases and can read a given text file, encapsulate the text into data segments, then send multiple of these data segments in a pipelined manner to the receiver. It can also emulate the behaviour of packet loss. It can also receive acknowledgements from the receiver.
- **Receiver.java** – The receiver receives data segments sent by the sender and processes them by writing the transferred bytes into a given text file. It also sends acknowledgements of all received data segments.

## PROGRAM DESIGN

### SENDER

The sender has three main threads:

- Main sending thread in **sendSegment() –** after being called, the method sits in a while loop until the program enters the CLOSED state. In the while loop, packets are synthesized differently into either SYN, DATA or FIN segments (determined by the program's state as specified in the spec). Based on a random generated float, the method will either drop or send the packet. If the packet is a DATA packet and is sent, the packet is added to a buffer for sent data in case it needs to be retransmitted. If the packet is the oldest unacknowledged segment, startTimer() is called to schedule retransmission once the timeout interval has passed without acknowledgement.
- **ReceiveThread()** – similarly, the thread sits in a while loop until the program enters the CLOSED state and allows the program to openly receive acknowledgements. After receiving acknowledgements, processAckSegment() is called and determines whether the ACK is expected or stale (received after timeout). If it is expected, the packet is removed from the sent data buffer, the sliding window is moved by one packet and one more DATA packet is allowed to be sent in sendSegment(). If stale, the thread ignores it and waits until the correct acknowledgement. If three duplicate ACKs are received, then handleTimeout() is called to retransmit the oldest unacknowledged packet in the sent data buffer.
- Timer thread in **startTimer()** – the thread creates a new asynchronous timer when called and schedules retransmission of the oldest unacknowledged packet in the sent data buffer after the timeout interval (by calling handleTimeout()).

### RECEIVER

The receiver has two main threads:

- Main receiving and responding thread in **receiveAndRespond()** - the thread sits in a loop until the program enters the CLOSED state. It receives incoming SYN, DATA and FIN segments, then sends a corresponding acknowledgement. If a DATA packet has the expected sequence number, a correct acknowledgement is sent and the data is immediately written to the text file. If a DATA packet is received out of order, the packet is added to a

receive buffer and only written to the text file when all packets before it are written to the file in-order. A duplicate ACK will be sent. Every time an expected DATA packet arrives, processBufferedPackets() is called to check if there are any buffered data packets which come sequentially after the expected packet, and writes them to the text file while updating the next expected sequence number.

- **TimerThread()** – this thread is only called if the program enters the TIME_WAIT state and sleeps for two seconds. If interrupted, it means that a FIN segment has been retransmitted and the **receiveAndRespond()** is still running and able to respond. If not interrupted, the program enters the CLOSED state and terminates.

## DATA STRUCTURE DESIGN

### STATES

Both sender and receiver each contain a static class of Enums of states, which are used throughout the programs to keep track of what type of segment to send and when to terminate the program.

#### SLIDING WINDOW/SENT DATA BUFFER

As previously mentioned, all sent DATA segments are added to a sent data buffer by the sender and only removed if the corresponding ACK is received. I chose to represent the sent data buffer in the form of Map<Integer, DatagramPacket>, where each sent packet is distinguished by its sequence number. In this way, the program is able to remove packets from the sent data buffer by simply removing based on the sequence number key, thereby making space available in the sent data buffer and allowing more packets to be sent.

The sliding window is technically not a data structure, since we determine that the packets are only able to be sent if the sent buffer's size is less than max_win / 1000. We determine if ACKs are expected if it meets the following condition:

```
(ackNo > oldestUnackedSeqNo && ackNo <= oldestUnackedSeqNo + max_win)
                || (ackNo < oldestUnackedSeqNo && ackNo <= (oldestUnackedSeqNo + max_win) % 65535).
```

We 'slide' the window by making the oldest unacknowledged sequence number equal to the last expected acknowledged number received, and by removing packets from the sent data buffer.

#### RECEIVED DATA BUFFER

As previously mentioned, any DATA segments received out-of-order by the receiver are placed into a receive buffer to wait for the correct DATA segment to be received. I chose to represent the receive buffer in the form of Map<Integer, byte[]>, where each sequence number is a key for the data bytes the packet holds. In this way, the program is able to lookup for a sequence number in the receive buffer equal to the recent expected acknowledgement number (since this is an indication that the DATA segment was received before the previous DATA segment) and is able to write its bytes to the text file in-order. This occurs in the processBufferedPackets() method.

## DESIGN TRADE-OFFS/FLAWS

I don't believe there were many design trade-offs caused by my design decisions. However, the use of Enums to represent the states meant that I inefficiently used many if statements within my code and caused my code to become harder to read/long. This could have been eliminated if a state design pattern was used. I don't believe there was any other way to represent the sliding window without the use of a hash map and arithmetic to calculate the next expected ACK, and I believe my hash maps allow for easier lookup of packets/byte arrays based on sequence number of the packet.