CS3103 Assignment 4 Report

Group: 46
Members:

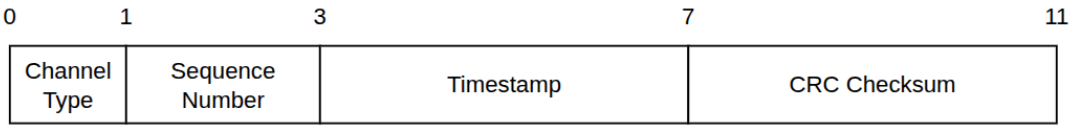
1. Jess Kwek Zhi Chen (A0292071J)
2. Sherisse Tan Jing Wen (A0254912E)
3. Tan Le Yew (A0272846U)

Github Repository URL: https://github.com/Blahblahlolhahaha/**GIB_A_PLS**_TT_3103

Design Choices

Packet Header

Our packet header is defined as follows:



The fields are:

- Channel Type
  - *Channel Type ∈ {0: RELIABLE, 1: UNRELIABLE, 2: ACK, 3: METRIC}*
- Sequence Number
  - modulo 2^16 sequence number
- Timestamp
  - sender's ms clock (lower 32 bits)
- CRC Checksum

Fragmentation and MTU

We assume payloads fit within path MTU for UDP; no in-protocol fragmentation.

GameNetAPI

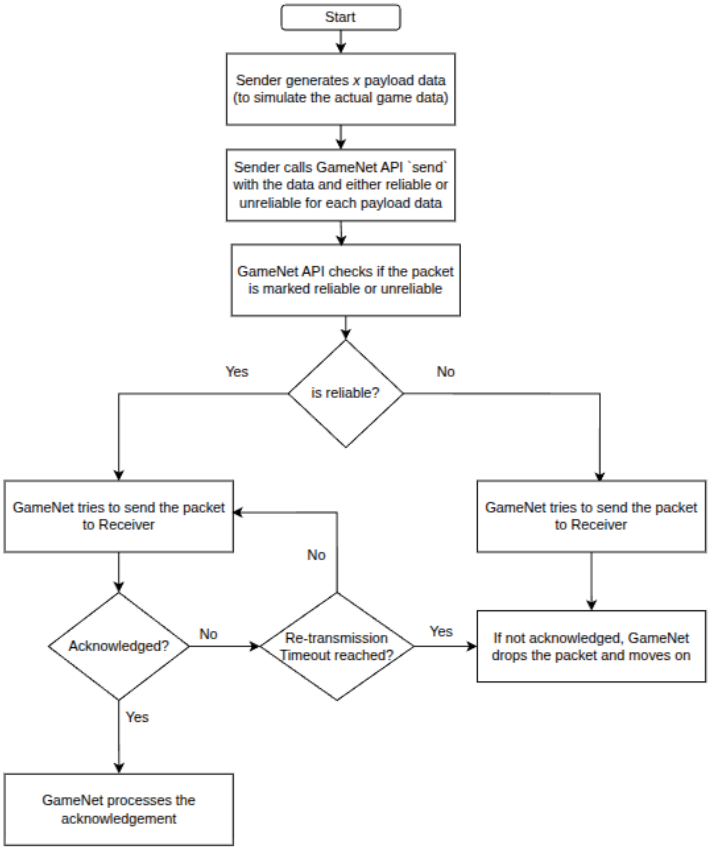
Goal: Two app channels on UDP: Reliable (0) and Unreliable (1); We use additional channels for ACK (2) and METRIC (3).
Reliable (Ch 0): Selective-repeat with map to keep track of packets that are pending ACK; ACKs (Ch 2) carry Seq and are consumed internally by GameNetAPI.

- `retransmission_timeout_ms` controls resend cadence
- `gap_skip_timeout_ms` lets us skip HOL after a timeout to keep the stream interactive. We validate `retx < gap_skip`. Per the assignment specifications, we use *200ms* by default.

Unreliable (Ch 1): Freshest-wins: deliver only if `(seq - last_seq) mod 2^16 ∈ (0, 2^15)`; older/stale packets are dropped. The half-modulo rule handles wraparound cleanly.
Metrics (Ch 3): On `close()`, sender sends counts; receiver computes Throughput, Avg Latency, Jitter, PDR and (optionally) logs CSV.
Threading: Receive (RX) thread (parse/CRC → route → ACK/send to app) + Retransmission (RETX) thread (timer-based resends). Uses small locks + queue for simple and predictable timing.

Why UDP (not QUIC): We implement retransmitting/ordering ourselves to learn trade-offs and QUIC would need modification to get such details.

Overall Program Flow



Sender will start by calling GameNet API `send` for each packet to be sent. For each packet, aside from randomly generating the payload, it also randomly marks each packet as either reliable or unreliable to denote which channel it should be sent over. The probability of marking a packet as unreliable is given by *reliable_ratio* which is defined when starting the sender.

GameNet then processes each packet, sending them over the appropriate channels. If the packet was marked reliable, GameNet attempts to retransmit the packet over the duration of the retransmission timeout until it is either successfully sent and thus acknowledged, or it is past the duration. GameNet will also reorder the packets if necessary. Otherwise, if the packet was marked unreliable, GameNet simply sends the packet without ensuring that it is acknowledged, and without reordering.

On the Receiver, for each packet that was received, it logs the Sequence number, channel type, send and receive timestamps, number of retransmissions as well as the estimated RTT (by taking 2x Latency) as per the assignment specifications.

Test Setup

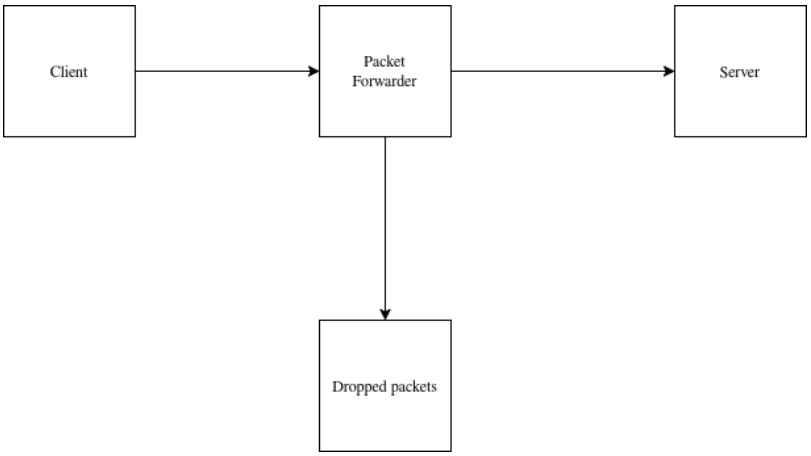
To simulate a real network with latency and delay, we decided to create a custom packet forwarder that forwards packets from the client to the server and vice versa. The packet forwarder accepts 3 arguments:

1. **P-LOSS**: A value from 0 - 1 which defines the probability the forwarder drops a packet which simulates packet loss.
2. **LISTENING-PORT**: The port which the forwarder will listen to packets from the client.
3. **SERVER_PORT**: The port at which the receiver is listening on.

The forwarder will also delay response for a random time between 0 - 40ms to simulate jittering/delay.

This allows us to precisely control various metrics as required and thus more easily introduce changes to the network, allowing us to test the entire system more rigorously. We also note that having this custom packet forwarder means that we have the capabilities we need in 1 application rather than potentially requiring more than 1 tool.

The following shows the network diagram for our setup, with the packet forwarder listening on port 8002 and the receiver listening on 8001.



The client accepts two parameters, one being the time in seconds to send and another one being the probability of the packet being set on the reliable channel while the receiver can accept a metric flag to set it to collect performance metrics!

Results and Analysis

For the analysis of the protocol, we test the protocol over 2 different loss environments:

- **Low loss** environment: 1% packet loss rate
- **High loss** environment: 20% packet loss rate

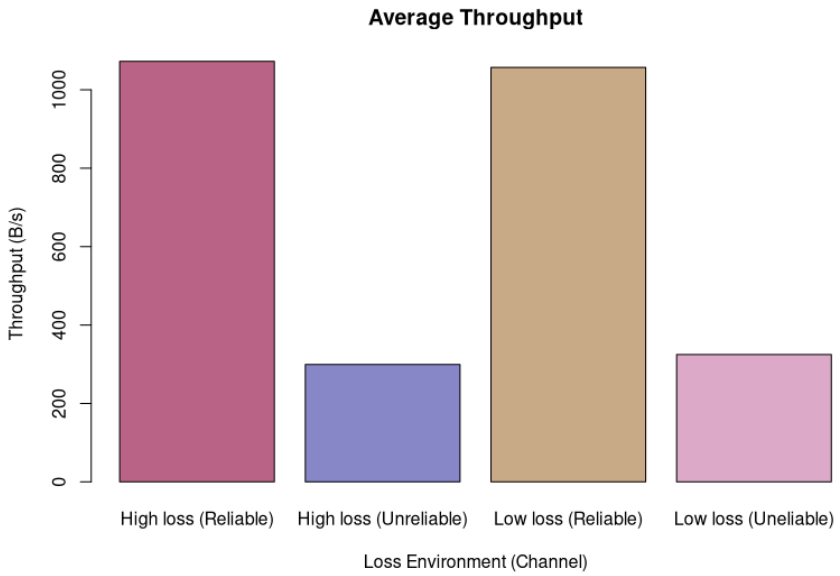
Here, we generated 30 packets/s to run over 3mins over the both loss environments to test the protocol comprehensively when receiving packets from both channels. Each packet has a 50% chance of being sent via the reliable channel and unreliable channel.

The gamenet api instantiated on the receiver side will be collecting metrics such as:

- **Latency** given by: epoch time upon receiving - timestamp of the packet, in milliseconds
- **Jitter** Given by J = (d - J)/16, where d is the difference in latency of successive packets in milliseconds.
- **Packet Delivery Ratio (PDR)**, given by: no. of packets received by receiver/no. of packets sent by sender.
- **Throughput**, given by: Data received/duration of the run

The information will then be saved into a csv file which will be used for analysis later. Information is collected separately for both reliable and unreliable channels, and differentiated by the channel column in the csv file. Each loss environment is tested separately and saved in different csv files. Below is a bar plot of all the collected variables across all channels and loss environments:

Throughput

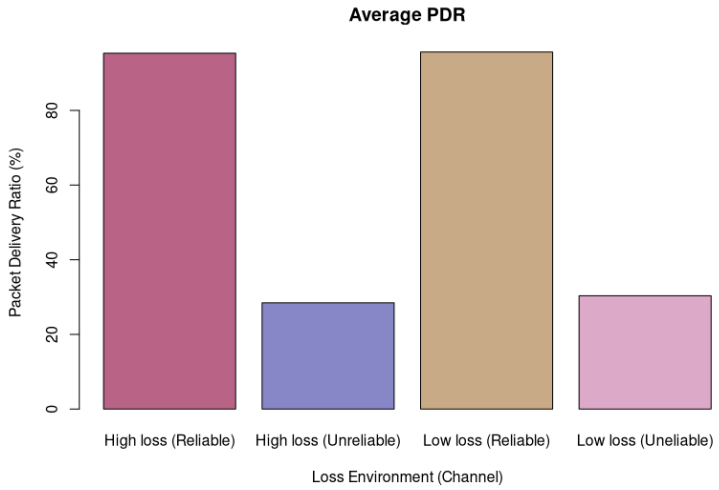


Data:

- High loss Throughput (reliable): 1072.54 bytes/s
- High loss Throughput (unreliable): 299.43 bytes/s
- Low loss Throughput (reliable): 1057.12 bytes/s
- Low loss Throughput (unreliable): 324.81 bytes/s

For the reliable channel, the throughput is higher on the low loss channel as lesser packets are lost, leading to fewer retransmissions required to transmit the packets, meaning it takes on average lesser time to send the same number of data, leading to higher throughput!

Packet Delivery Ratio (PDR)



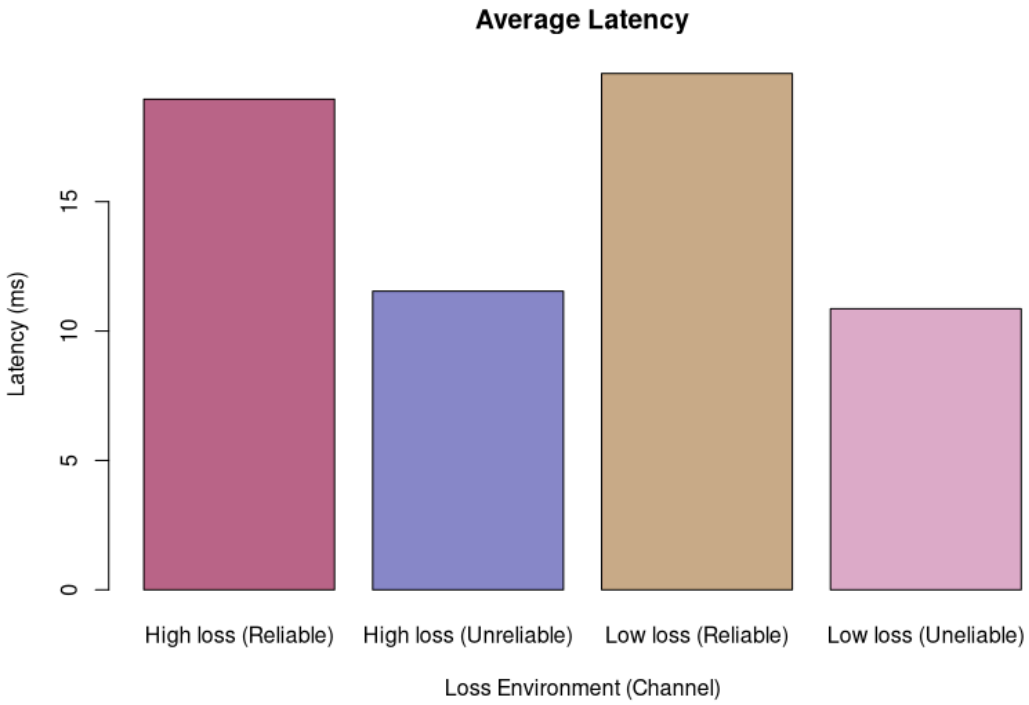
Data:

- High loss PDR (reliable): 95.31%

- High loss PDR (unreliable): 28.47%
- Low loss PDR (reliable): 95.64%
- Low loss PDR (unreliable): 30.35%

For PDR, the reliable channel delivery ratio remains high for both high loss and low loss environments as the channel tries to ensure packet delivery in the case of packet loss. While for the unreliable channel, the differing delay for each packet causes some newer packets to reach the receiver faster than the older ones, causing all the older packets to be dropped as the unreliable channel does not guarantee that the packets arrive in order.

Latency

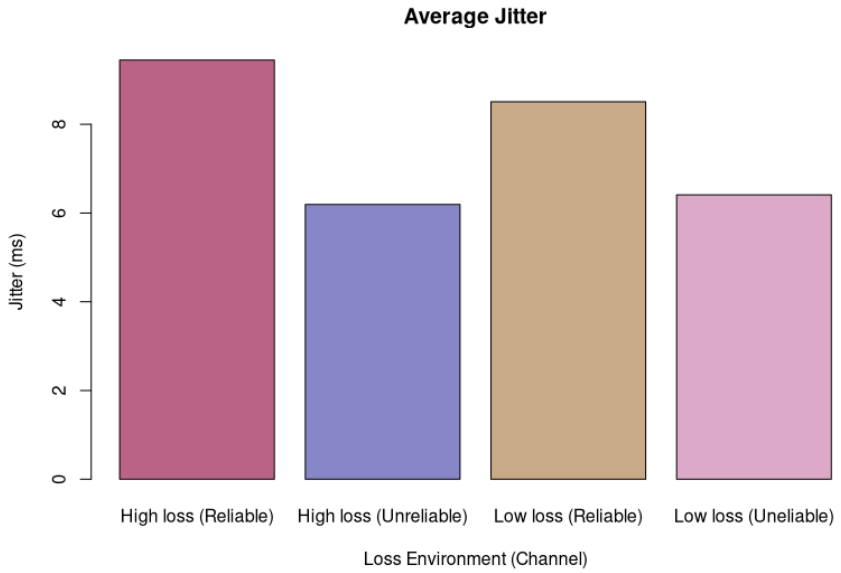


Data:

- High loss latency (reliable): 18.96ms
- High loss latency (unreliable): 11.54ms
- Low loss latency (reliable): 19.95ms
- Low loss latency (unreliable): 10.86ms

The reliable channel experiences a higher latency as compared to the unreliable channel as there is more overhead when waiting for acknowledgement and for the retransmission of any packets that are not acknowledged, leading to a higher average latency as compared to the unreliable channel, which does not require acknowledgement from the server and thus does not wait.
The reliable channel also buffers out of order packets to wait for all packets to arrive in sequence whereas the unreliable channel does not and will instead immediately send the packet on arrival.

Jitter



Data:

- High loss jitter (reliable): 9.45ms
- High loss jitter (unreliable): 6.19ms
- Low loss jitter (reliable): 8.51ms
- Low loss jitter (unreliable): 6.41ms

It can be noted that the both unreliable and reliable channels are similar in jitter values since in the experiment, the difference between 2 successive packets is mostly due to the artificial delay introduced by our packet forwarder, making delays between the arrival of each packet from the reliable and unreliable channels similar! It can be noted that jitter is unaffected by any packets dropped or the actual latency of each packet, but measures the stability of the connection!

Reflections

Jess

From this assignment, I was able to appreciate the process behind designing protocols and the various considerations that goes behind each decision made when designing the various aspects of how to ensure reliable data transfer via acknowledgement of the packets/retransmission after a certain amount of time and reordering out of order packets how does unreliable data transfer work after an unreliable packet is received.

On top of that, I was able to use the knowledge I gained from statistics this sem to help collect the metrics and analyse the metrics across different environments and how to interpret the information collected to and figure out the underlying reasons such as the reliability of the channel affecting the packet delivery ratio. This allowed me to connect different modules together and let me see how the concepts taught in various disciplines can connect with each other!

Sherisse

Through this assignment, I gained a better understanding of transport layer protocols, as well as the various trade-offs and considerations that come into play when designing one of our own.

Specifically, the protocol design depends heavily on what application we are trying to create, in terms of whether we can accept unreliability in our packet transmissions such as lost / corrupted / reordered packets, or whether we require reliability in terms of making efforts to retransmit packets. If we are able to accept this trade-off, then we can potentially increase the speed of our data transmission as compared to if we had to buffer packets / retransmit them, etc. when ensuring reliable delivery.

Developing the two end systems required me to understand the whole end-to-end process of sending data, as well as receiving and processing data, and what happens in between (in the GameNet API). This gave me a better understanding of the whole process, and what is actually happening when we send data over the network.

Le Yew

This assignment showed me how tricky it is to juggle multiple channels, buffers, and edge cases as it was easy to make small mistakes when coding, even if specs are already well defined. Using LLMs helped me to accelerate ideas, such as brainstorm buffer sizes, sketch Go-Back-N vs. Selective Repeat skeletons, break down RFCs, and think through why implementing this in QUIC may be non-trivial.

Overall, my biggest takeaway is that transport design is a constant trade-off: speed vs. data reliability vs. implementation complexity. Making the “right” trade-off “right” choice depends on the use case and the time you have to build (with more time, you can add things like exponential backoff, ACK piggybacking, better batching).

Team Level

When designing the transport layer protocol design, we have to consider trade-offs in terms of whether we are able to accept lost / reordered / corrupted packets or whether we require packets to always be delivered if possible, i.e., attempt retransmissions *x* times total, or for a period of time.

If we can accept unreliability in the packet transmission, we can then send data faster with greater throughput. However, if we need reliability, i.e., we need the packets to always be transmitted if possible and thus we attempt retransmissions, then this will result in a corresponding decrease in throughput and greater latency.

From the results we collected, it shows that the reliable channel should be used for critical data that is required to be sent reliably without it being lost in transmission such as asset downloading and game results while the unreliable channel should be used when speed is more important than the reliable transfer of data such as voice chat.

Through this assignment, we have learned more about designing transport layer protocols from scratch, building on top of UDP directly. We also learned more about multi-threading as well as creating libraries, and APIs, that other applications can import and utilise easily without much overhead. From the process of creating our own packet forwarder to simulate various network conditions, we also gain more knowledge and an in-depth understanding of the entire system and process.

Program Instructions

We have 4 different main applications:

1. `gamenet_api.py`
2. `sender.py`
   - `python3 sender.py <num_secs> <reliable_ratio>`
3. `receiver.py`
   - `python3 receiver.py` (`-m` flag to enable metrics mode)
4. `unrelinet.py`
   - `python3 unrelinet.py <p_loss> <listen_port> <server_port>`

To simulate real-world network conditions: run `unrelinet` first before the other applications
Otherwise:

1. Run `receiver.py`
2. Run `sender.py`

There is no need to run GameNet manually as it is imported and started by both sender and receiver automatically.

Acknowledgements

When designing `unrelinet.py` to simulate the real-world network conditions, we took inspiration and reference from the `unrelinet.java` application that was provided as part of an assignment in CS2105 in AY24/25 Semester 2.
This original `unrelinet.java` file can be found in `reference/UnreliNET.java` in the github repo / the zip file submitted.