

CS3103 Assignment 4 Technical report

GameNetApi	2
GameNetApi design	2
Assumption	2
Overall protocol design	2
Receiver API design	3
Sender API design	4
Test	5
Test setup	5
Test result	6
Perfect Network	6
Mildly Unstable Network (Wi-Fi / LAN with background traffic)	7
Moderate Unstable Network (Mobile / Wireless Network)	7
Highly Unstable Network (Severe Congestion)	8
Other Test setup	9
Reflection	10
Acknowledgement	10

GameNetApi

[Repo](#)

[Demo video](#)

GameNetApi design

Assumption

Before we discuss the details, there are a few assumptions to ensure proper usage.

- **Sender** refers to the user of our sender API, and **Receiver** refers to the user of our receiver API.
- We assume the data transmission is **unidirectional**, meaning only the sender transmits “important” data to the receiver. The receiver only sends ACKs to acknowledge the progress of received data.
- We also assume that data transmissions are **non-interleaving**, meaning reliable and unreliable data will not be mixed during reception. However, we do provide APIs for both reliable and unreliable channels.

The overall flow for using our API is as follows (this will also help when understanding the design motivation later):

- **Sender:** Has multiple small data chunks to send → decides whether to use the reliable or unreliable channel → calls the appropriate API.
Receiver: Creates the receiver server → provides a callback function to handle incoming data → waits until all data are received → retrieves the data (reliable channel only).

We have provided **receiverExample.py** and **senderExample.py** to demonstrate the correct way to use the APIs.

Overall protocol design

For this protocol, we have 2 main parts, which are **Sender** and **Receiver**. We have designed 2 different headers for these 2 parts, which serve for different purposes.

- **Sender:**
 - The header format is:

Channel (8 bits)	Flags (8 bits)	Sequence Number (16 bits)	Payload Length (16 bits)	CRC Checksum (32 bits)	Timestamp (32 bits)
---------------------	-------------------	---------------------------------	--------------------------------	------------------------------	------------------------

- **Channel (8 bits)** : Indicates the packet type — “0” for an **unreliable** packet and “1” for a **reliable** packet. Reserved bits are kept for future extensions.

- **Flags (8 bit)** : Indicates whether the packet is the **last packet** in a sequence — “1” for the last packet, “0” otherwise. Reserved bits are kept for future extensions.
- **Sequence Number (16 bits)** : Identifies the order of the current packet, allowing the receiver to **reassemble packets correctly** and determine the appropriate **acknowledgment number** to send.
- **Payload Length (16 bits)** : Specifies the size of the payload, enabling the receiver to **measure connection throughput** and manage data processing efficiently.
- **CRC Checksum (32 bits)** : Contains the computed checksum of the packet, allowing the receiver to **verify data integrity** and **detect any corruption** during transmission.
- **Timestamp (32 bits)** : Records the sending time of the current packet, allowing the sender to **calculate the Round-Trip Time (RTT)** for performance analysis.

- **Receiver**

- For the receiver, we only send ACK for reliable packets. For an unreliable packet, we just receive it and store it into the buffer.
- The header format is:

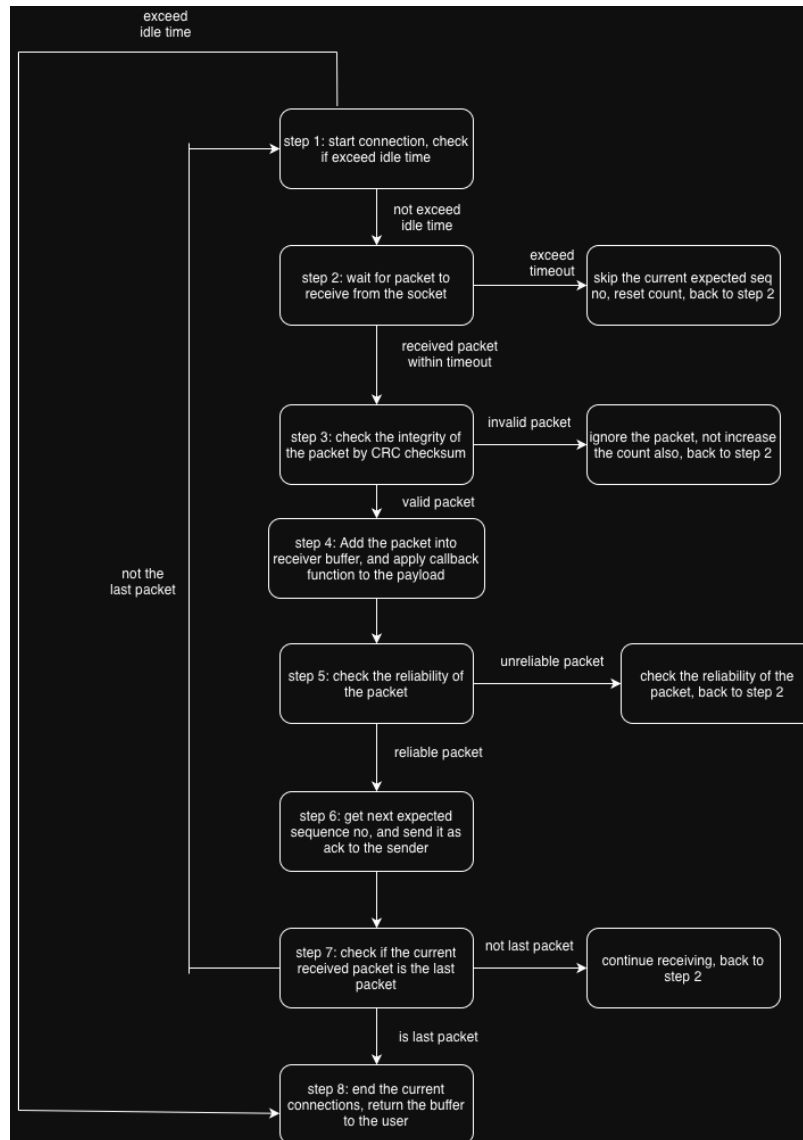
Flags (8 bits)	ACK sequence number (16 bits)	CRC Checksum (32 bits)	Timestamp (32 bits)
---------------------	---------------------------------------	-----------------------------	--------------------------

- **Flags (8 bits)** : Indicates whether the sender should **continue or stop** transmitting packets — “0” to continue, “1” to stop. Reserved bits are kept for future extensions.
- **ACK sequence number (16 bits)** : Represents the acknowledgment number from the receiver, defined as **the highest sequence number received + 1**. It indicates the **next expected packet** from the sender.
- **CRC Checksum (32 bits)** : Contains the computed checksum of the packet, allowing the receiver to **verify data integrity** and **detect any corruption** during transmission.
- **Timestamp (32 bits)** : Records the **timestamp of the most recently received packet**, enabling the sender to **calculate the Round-Trip Time (RTT)** for connection performance monitoring.

Receiver API design

We offer a **general receive function** where the user can provide a **callback** to process incoming data. Once the connection ends, the function **returns all received data** to the user.

The workflow of receiver API is as following:

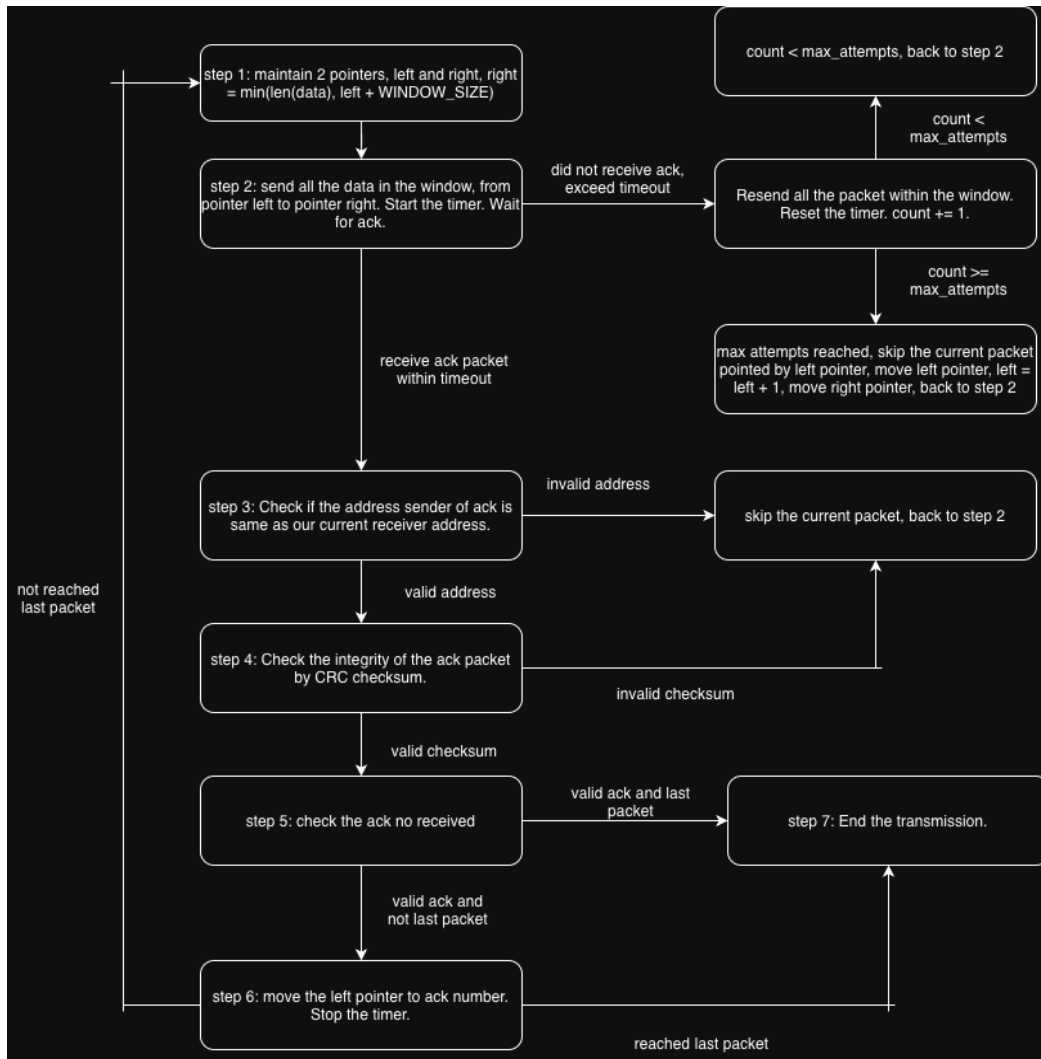


Sender API design

We provide a **general send function** that allows the user to pass a **list of data** along with a **flag indicating reliable or unreliable delivery**. The function handles each type separately:

- **Unreliable sending:** Data is transmitted **sequentially over a raw UDP socket** without additional checks or retransmissions.
- **Reliable sending:** Data is transmitted using the **Go-Back-N protocol**, ensuring **ordered delivery** and **automatic retransmission** of lost packets. A **timer** tracks each packet's timeout; if a packet exceeds the timeout, it is **skipped** and the sender proceeds with the next packet.

The workflow for the sender reliable sending API is as following:



Test

We use Docker to ensure compatibility across different operating systems. It also allows us to simulate various network conditions between the sender and receiver using tc-netem. Since everything is containerized, the Python version is not an issue.

Our README.md provides the basic command flow for setup. In short, after running `docker compose build`, you can simply run `server.sh`, followed by `sender.sh`. Change the parameter of tc-netem in docker-compose.yml to simulate different network condition

Test setup

For consistency in testing, we kept the configuration parameters unchanged, the most important thing is the window size is 10. The test input is a list of data containing over 900 small packets, with a total size of 50,000 bytes.

The following network conditions were simulated:

1. **Perfect Network:** No packet loss, 10 ms delay (to prevent network overload).

2. **Mildly Unstable Network:** 100 ms delay, 10 ms jitter, 1 % packet loss.
3. **Moderately Unstable Network:** 150 ms delay, 25 ms jitter, 1% loss, 0.5 % reordering (25 % correlation)
4. **Highly Unstable Network (Severe Congestion / Long-Distance Connection):** 200 ms delay, 20 ms jitter, 15 % packet loss, 1 % reordering (25 % correlation)

Test result

Perfect Network

Unreliable	Reliable
min / mean / max jitter :0.0 / 0.3207 / 0.6472 throughput :232547.9452 / 7508625.4952 / 807000000.0 latency :0 / 0.1956 / 1 packets received :946 expected total packets :946 packet delivery ratio :1.0	min / mean / max jitter :0.25 / 0.9803 / 1.4872 throughput :9523.8095 / 22086.6498 / 24125.7576 latency :0 / 2.1526 / 6 packets received :946 expected total packets :946 packet delivery ratio :1.0

We set the packet-sending interval to **10 ms** to avoid overwhelming the network interface, as sending too many packets simultaneously could lead to unintended packet drops. During testing, we also observed that for **unreliable transmissions**, adding a small `time.sleep()` delay was necessary to prevent buffer overflow at the network interface, which otherwise resulted in packet loss.

Nevertheless, the recorded data represents the **baseline performance** achievable under ideal network conditions. These results show the **maximum throughput** that the system can sustain without causing congestion or loss, establishing a clear reference point for comparison with other, less stable network environments.

Mildly Unstable Network (Wi-Fi / LAN with background traffic)

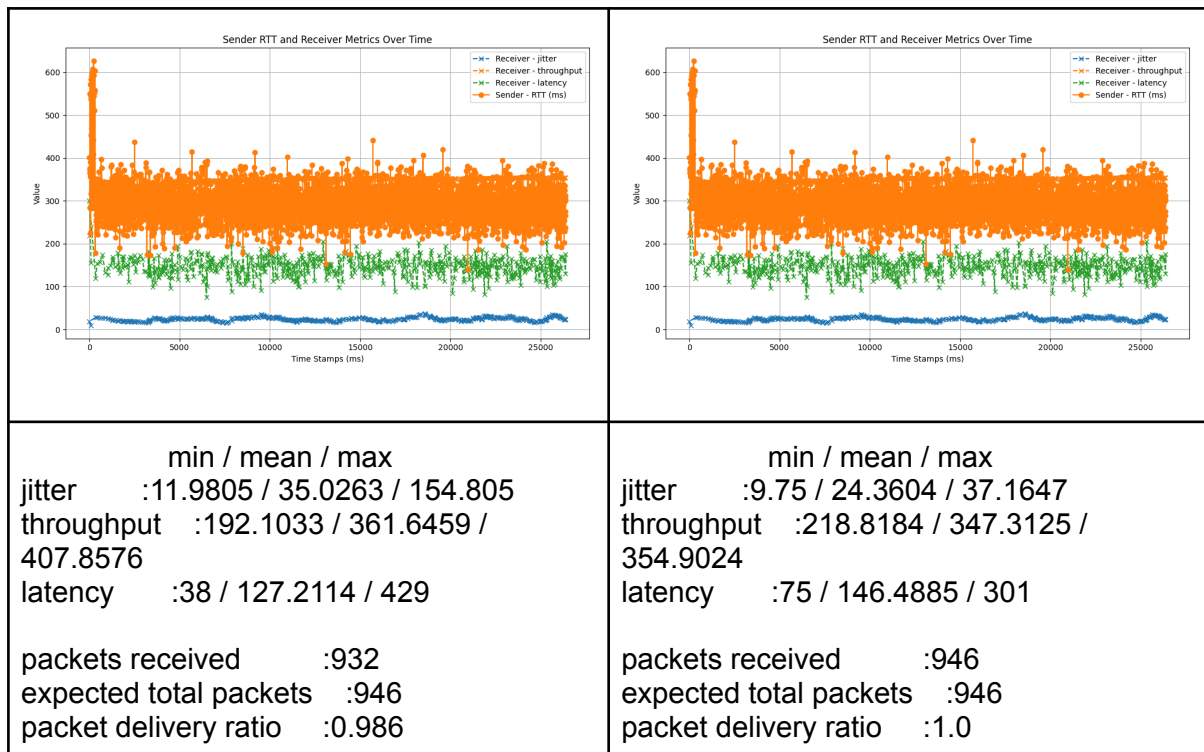
Unreliable	Reliable
min / mean / max jitter :6.0625 / 12.9947 / 21.4358 throughput :486.618 / 510.2246 / 534.7594 latency :74 / 101.5135 / 137 packets received :929 expected total packets :939 packet delivery ratio :0.9894	min / mean / max jitter :7.2264 / 11.1737 / 30.1211 throughput :164.4737 / 500.9058 / 515.8871 latency :71 / 100.9538 / 304 packets received :946 expected total packets :946 packet delivery ratio :1.0

After the network stabilized, we observed for unreliable transmissions, the **RTT** was approximately twice the average **latency**, which is expected because it includes both the forward and return paths.

The **packet delivery ratio** for the unreliable channel closely matched the configured **1% loss rate**, while the reliable channel achieved **full delivery** through retransmissions. However, the **throughput** decreased noticeably in both cases compared to the ideal network. Despite this reduction, the reliable channel still performed well, successfully maintaining data delivery with only a slight increase in delay and minimal packet loss.

Moderate Unstable Network (Mobile / Wireless Network)

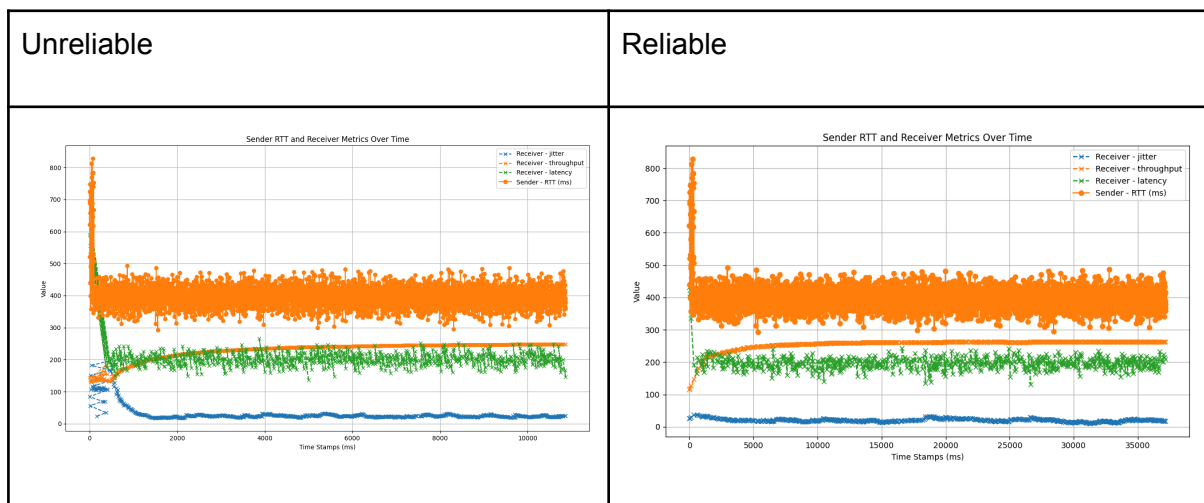
Unreliable	Reliable
------------	----------



Examining the server logs, we observed that there were no skipped packets under this condition, but the transmission time was noticeably longer compared to previous tests, indicating the impact of a more unstable network. From the collected data, we can see that the **throughput decreased further** as the network conditions deteriorated. The increased **delay**, **jitter**, and occasional **packet reorder** reduced the overall sending rate for both reliable and unreliable transmissions.

Additionally, the **reliable protocol** experienced a further drop in throughput due to higher latency, increased jitter, and packet reordering in the network. Despite these challenges, it successfully delivered **all packets in order**, demonstrating strong robustness in maintaining **data completeness and sequence integrity** under moderately unstable network conditions.

Highly Unstable Network (Severe Congestion)



	min / mean / max		min / mean / max
jitter	:5.3125 / 29.8678 / 42.0919	jitter	:10.7469 / 20.6003 / 37.0463
throughput	:343.2789 / 346.5681 / 588.2353	throughput	:115.7407 / 256.2225 / 263.2992
latency	:74 / 150.6013 / 223	latency	:131 / 197.3661 / 432
packets received	:923	packets received	:946
expected total packets	:937	expected total packets	:946
packet delivery ratio	:0.9851	packet delivery ratio	:1.0

```

receiver-1 | Waiting for packet 159 remaining time: 0.01 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 152, 52, 1665591691, 315268639)
receiver-1 | Waiting for packet 159 remaining time: 0.01 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 160, 52, 3234681078, 315268639)
receiver-1 | Waiting for packet 159 remaining time: 0.01 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 153, 52, 3041341199, 315268679)
receiver-1 | Waiting for packet 159 remaining time: 0.01 s
receiver-1 | Exceeded time, skip the packet.
receiver-1 | Waiting for packet 160 remaining time: 0.20 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 157, 52, 2869048631, 315268679)
receiver-1 | Receive the data: b'161: This is a sample line of text to fill the file.'
receiver-1 | Waiting for packet 161 remaining time: 0.20 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 161, 52, 356731700, 315268678)
receiver-1 | [Reliable] (1, 0, 161, 52, 356731700, 315268678)
receiver-1 | Latency: 182 ms
receiver-1 | Throughput: 262.68 bytes/s
receiver-1 | Jitter: 17.42 ms
receiver-1 |
receiver-1 | Receive the data: b'162: This is a sample line of text to fill the file.'
receiver-1 | Waiting for packet 162 remaining time: 0.20 s
receiver-1 | Received packet from ('192.168.107.3', 58467): (1, 0, 159, 52, 1153100853, 315268625)

```

From the server logs, we observed that several packets were skipped, and many arrived out of order. However, the data still performed well because our receiver buffers all incoming packets, including unexpected ones. This design allows delayed packets that arrive after being skipped to still be included in the final reconstructed data.

This result highlights the strength of our **partial reliability mechanism**. Compared to the previous network condition, the overall quality of the connection was much worse, yet the reliable channel required only about **10 additional seconds** to complete transmission. It maintained a **rather good packet delivery ratio** with only a minor reduction in throughput due to retransmissions. This trade-off demonstrates the protocol's effectiveness in ensuring **data completeness and consistency** even under severely congested and unstable network conditions.

Other Test setup

While testing our API, we found that a **200 ms skip threshold** provides the best balance between responsiveness and reliability. If the threshold is set too low, the server skips packets too quickly, which negatively affects reliability. Conversely, if it is set too high, the receiver waits too long for missing packets, leading to higher overall transmission delay. Hence, 200 ms is an optimal value that maintains smooth performance without significant delay.

For the **sender window size**, there is still room for improvement by increasing it. However, a larger window size also introduces additional overhead during retransmission, as a wider range of packets needs to be resent. Based on our experiments, we found that using a window size of 10 strikes a good balance between performance and efficiency.

Reflection

Reflection on Design Trade-offs:

1. Our receiver buffer uses **16 bits** for the sequence number, meaning it can handle up to **2^{16} packets per transmission burst**. Therefore, the sender must either include more data in each packet or send fewer packets within a single burst.
2. Currently, the sender cannot use our API to send **reliable and unreliable packets simultaneously**, as the receiver enforces an **idle waiting period** to collect incoming packets. During this idle time, users may still send additional packets over the same channel, but not across different reliability modes. This idle period can temporarily block the sender from transmitting other types of packets until the receiver finishes processing the current batch.
3. We intentionally **removed the TCP-style three-way handshake** to reduce connection setup time and improve responsiveness. Consequently, the sender must ensure that the receiver has already invoked the `receive_data()` function before transmission, as our API does not include an explicit reachability confirmation between endpoints.
4. On the sender side, we implemented a mechanism similar to the **Go-Back-N (GBN)** protocol. When a timeout occurs, the sender **retransmits all unacknowledged packets**, even if some have already been received successfully, which introduces some inefficiency and potential bandwidth waste.

In conclusion, designing a hybrid transport protocol inherently involves balancing latency, reliability, and complexity. Each design choice, such as whether to include connection establishment, how to handle retransmissions, or how to manage packet buffering affects both performance and usability. Through this project, we learned that there is no single “best” configuration; rather, the effectiveness of a protocol depends on its intended use case and network conditions. Our final design reflects a practical compromise, focusing on maintaining reliability while minimizing delay, making it suitable for real-time applications such as online games.

Acknowledgement

We use ChatGPT to find bugs, fix document grammatical errors. We are responsible for our work.