CS3103 Assignment Technical Report

1. Design Choice

- **gameNetAPI:** reliable UDP with Selective Repeat; unreliable UDP
- **packet design:** see below
- **ack:** SACK (Selective Acknowledgment)
- **synchronization:** concurrent access is synchronized to prevent race conditions
- **skipping rule:** default set to 200 ms; configurable

2. GameNetAPI

Class: `ReliableUDP_API`

Constructor

```
ReliableUDP_API(local_port: int, remote_host: str | None = None, remote_port: int
| None = None)
```

Description

Creates a new Reliable UDP endpoint.
If `remote_host` and `remote_port` are specified, the instance can send messages; otherwise, it acts as a **receiver-only** endpoint.

Parameters

Name	Type	Description
`local_port`	`int`	Local UDP port to bind for listening.
`remote_host`	`str`, optional	Remote peer IP or hostname.
`remote_port`	`int`, optional	Remote peer UDP port.

Example

```
receiver = ReliableUDP_API(local_port=8000)

sender = ReliableUDP_API(local_port=9000, remote_host="127.0.0.1",
remote_port=8000)
```

Method: `send`

```
send(data: bytes, reliable: bool = True) -> None
```

Description

Sends a message to the remote peer.

- When `reliable=True` (default): uses **reliable channel** with acknowledgments, retransmissions, and ordered delivery.
- When `reliable=False`: uses **unreliable channel**, which sends the packet once without guarantees.

Parameters

Name	Type	Default	Description
`data`	`bytes`	—	Payload to send.
`reliable`	`bool`	`True`	Whether to send via reliable channel.

Raises

- `RuntimeError`: if called on a receiver-only endpoint (no remote address).

Example

```
# Reliable message (default)
sender.send(b"Hello, reliable world!")

# Unreliable message
sender.send(b"Quick update", reliable=False)
```

Method: `receive`

```
receive() -> tuple[int | None, int, bytes] | None
```

Description

Retrieves the next available message from the internal delivery queue.
This method is **non-blocking** — it returns immediately if no data is available.

Returns

Type	Description

Type	Description
`(seq, ts_ms, payload)`	For reliable packets — includes sequence number and timestamp.
`(None, ts_ms, payload)`	For unreliable packets — sequence number is `None`.
`None`	If no packets are available.

Example

```
msg = receiver.receive()
if msg is not None:
    seq, ts, data = msg
    print(f"Received: {data} (seq={seq}, ts={ts})")
```

Method: `close`

```
close() -> None
```

Description

Closes the ReliableUDP_API instance, stops background threads, cancels timers, and closes the underlying UDP socket.

Example

```
sender.close()
receiver.close()
```

3. Test Setup

The testing environment is automated using a Bash script (`test.sh`) that simulates real network behavior and verifies both reliable and unreliable transmission over UDP.

Overview

The script launches:

1. A **receiver** process running `ReliableUDP_API` in background.
2. Optionally applies **`tc netem`** on the loopback interface to emulate network conditions such as loss, delay, jitter, and packet reordering.
3. A **sender** process that transmits both reliable and unreliable packets interleaved at a specified rate.

Configuration Parameters

All parameters can be customized via environment variables before running the script.

Variable	Default	Description
`RPORT`	6000	Receiver port.
`SPORT`	6001	Sender local port.
`REMOTE_HOST`	127.0.0.1	Target host address.
`REMOTE_PORT`	`$RPORT`	Destination port for sender.
`USE_NETEM`	1	Enable (`1`) or disable (`0`) `tc netem`.
`LOSS`	10	Packet loss percentage.
`DELAY_MS`	200	Mean network delay (ms).
`JITTER_MS`	50	Jitter (ms).
`REORDER`	20	Packet reorder percentage.
`NREL`	50	Number of reliable packets to send.
`NUNREL`	20	Number of unreliable packets to send.
`PPS`	10	Packets per second.
`DURATION`	20	Receiver runtime in seconds.

Test Cases

To validate the functionality and robustness of the Reliable UDP stack, several test cases were designed and executed using the `test.sh` framework.

1 Functional Correctness

Objective: Verify correct operation of reliable and unreliable channels under normal conditions.
Setup:

- No packet loss or delay (`USE_NETEM=0`).
- Sender transmits 20 reliable and 10 unreliable packets.
  Expected Outcome:
- All reliable packets are received in order.
- Unreliable packets are received immediately but may be interleaved.
- No retransmissions or missing packets are observed.

2 Packet Loss Simulation

Objective: Test retransmission and Selective Repeat (SR) reliability.
Setup:

- Enable `LOSS=10`.
- 50 reliable and 20 unreliable packets.
  Expected Outcome:

- Lost reliable packets are retransmitted until acknowledged.
- Receiver eventually delivers all reliable packets in order.
- Some unreliable packets are permanently lost.

3 Delay and Jitter

Objective: Assess timing tolerance and in-order buffering under variable latency.
Setup:

- Apply `DELAY_MS=200`, `JITTER_MS=50`, `LOSS=0`.
- 40 reliable and 20 unreliable packets at 10 pps.
  Expected Outcome:
- Receiver handles delayed packets with reordering buffer.
- Delivery latency increases, but sequence integrity is preserved.
- Unreliable packets arrive with non-deterministic delay.

4 Packet Reordering

Objective: Verify the receiver's buffering and skip-deadline mechanism.
Setup:

- Apply `REORDER=20` with default delay.
- Skip deadline set to 200 ms.
  Expected Outcome:
- Receiver buffers out-of-order packets and delivers them after missing ones or after timeout.
- Logs show skip events when gaps persist beyond deadline.

5 High-Load Stress Test

Objective: Evaluate stability under high packet rates.
Setup:

- `NREL=80`, `NUNREL=20`, `PPS=80`.
- Network emulation disabled (`USE_NETEM=0`).
  Expected Outcome:
- All reliable packets delivered correctly without congestion collapse.
- Receiver prints a continuous sequence of mixed reliable/unreliable packets.

Test result

1 Partial sample output

```
[STEP] starting receiver on 0.0.0.0:6000 ...
API (Receiver) listening on ('0.0.0.0', 6000)
[Receiver] up on :6000
[STEP] applying netem on lo: loss=10% delay=200ms ±50ms reorder=20%
[STEP] starting sender from :6001 -> 127.0.0.1:6000
API (Receiver) listening on ('0.0.0.0', 6001)
API (Sender) bound to ('0.0.0.0', 6001), sending to ('127.0.0.1', 6000)
```

```
[Sender] up, sending reliable + unreliable
API (Sender) RETRANSMIT: Seq 0 timed out. Resending.
[Receiver] U seq=- len=3 payload=b'U-0'
API (Sender) RETRANSMIT: Seq 1 timed out. Resending.
API (Sender) RETRANSMIT: Seq 0 timed out. Resending.
...
...
[Receiver] R seq=49 len=4 payload=b'R-49'
API (Sender) RETRANSMIT: Seq 49 timed out. Resending.
API (Sender) RETRANSMIT: Seq 48 timed out. Resending.
API (Sender) RETRANSMIT: Seq 49 timed out. Resending.
API (Sender) RETRANSMIT: Seq 48 timed out. Resending.
API (Sender) Received ACK for 49
API (Sender) RETRANSMIT: Seq 48 timed out. Resending.
API (Sender) RETRANSMIT: Seq 48 timed out. Resending.
API (Sender) Received ACK for 48
Closing API... stopping threads...
API closed.
[Sender] finished: reliable=50, unreliable=20
        Closing API... stopping threads...
API closed.
[Receiver] closed
[ALL DONE] test finished.
[CLEANUP] killing receiver (pid=47692) ...
[CLEANUP] removing tc qdisc from lo ...
[CLEANUP] done.
```

2 result for each case

- Functional Correctness: passed
- Packet Loss Simulation: passed
- 3 Delay and Jitter: passed
- 4 Packet Reordering: passed
- 5 High-Load Stress Test: passed

4. Packet Header Format

```
+-----------+---------------------+----------------------------------+
| 1 byte    | 2 bytes             | 4 bytes                          |
+-----------+---------------------+----------------------------------+
| chan      | seq                 | ts_ms                            |
+-----------+---------------------+----------------------------------+
```

5. Protocol Flowchart

[Sender]

- S0 [Application calls send(data, reliable)]
  S1 {channel type?}

- Reliable
  S2 [Build packet: DATA_CHANNEL seq=rsn, ts=now]
  S3 [Send pkt(DATA)]
  S4 [Start timer T_rsn]
  S5 [Store send_buffer [rsn]=pkt]
  S6 [Increment rsn and inflight counter]

- Unreliable
  SU2 [Build packet: UNREL_CHANNEL useq, ts=now]
  SU3 [Send pkt(UNREL)]
  SU4 [Increment useq]

- Receive ack
  S7 {{Receive ACK(ack_num)}}
  S8 [Cancel timer T_ack_num]
  S9 [Remove ack_num from send_buffer]
  S10 [Free window slot → send from pending queue]
  S11 {{Timer T_x expired?}}
  S12 [Retransmit]
  S13 [Restart timer for x]
  end

[Receiver]

- R0{{Incoming packet?}}
  R1 {Channel type?}
  R2 [Unpack header: (chan, seq/-, ts, payload)]

- Reliable Channel

  R3 [Send ACK(seq)]
  R4 {seq < next_expected?}
  R5 [Ignore duplicate packet]
  R6 [Cache receive_buffer[seq]=payload,ts]
  R7 {{Continuous seq from next_expected?}}
  R8 [Deliver to app queue: (seq, ts, payload)]
  R9 [Increment next_expected]
  R10 [If progressed → clear skip_deadline]
  R11 {Any gap remaining?}
  R12 [If not set → set skip_deadline = now + SKIP_TIMEOUT]

- Unreliable Channel

  RU1 [Deliver directly:(None, ts, payload)]

- Skipping

  RIdle {{Socket timeout (on_idle)}}
  RSk {{Skip deadline reached and gap still missing?}}
  RJmp [Skip missing seq: next_expected++]
  RClr [Clear skip_deadline]
  RTry [Try to deliver continuous packets again]
  Reset(If new gap remains)
  end

6. Latency/Jitter/Throughput

run the test in windows 1.37.0.0, with receiver and sender attached to local host, port 6000 6001

Latency:

- Reliable Channel Latency:
  Average: 0.20 ms
  Min: 0.00 ms
  Max: 1.00 ms

- Unreliable Channel Latency:
  Average: 0.19 ms
  Min: 0.00 ms
  Max: 1.00 ms

Jitter:

- Unreliable Channel Jitter (StdDev): 0.39 ms
- Reliable Channel Jitter (StdDev): 0.40 ms

Throughput:

- Total Bytes Received: 3895119
  Total Test Duration: 35.01 s
  Average Throughput: 890.00 kbps