RDT Protocol

Release 1.0

Geoffrey Tse

Oct 31, 2025

CONTENTS:

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

HW3

1.1 client module

client.py

RDT client for GET/PUT via the emulator.

This client sends application-layer commands over a reliable data transfer session (RDTSession), which itself runs over UDP. The packets are routed through the network emulator, allowing tests of loss, corruption, duplication, and reordering.

Usage examples:
python client.py –server 127.0.0.1:12000 –emulator 127.0.0.1:10000 GET test.txt python client.py –server 127.0.0.1:12000 –emulator 127.0.0.1:10000 PUT sample.bin

Behavior summary:

- GET <file> - request a file from server, store locally as download_<file>
- PUT <file> - upload a local file to the server

`client.main()`

Entry point for the RDT file client.

Steps:

1. Parse command-line arguments.
2. Bind a local UDP socket.
3. Create an RDTSession that sends to the emulator (NOT directly to server).
4. Run a background thread to pump raw UDP packets into the session.
5. Execute either GET or PUT.
6. Shutdown cleanly, ensuring retransmission thread exits before closing socket.

`client.win_udp_no_connreset(`*sock: socket*`)` → None

Disable WSAECONNRESET behavior on Windows UDP sockets.

When a UDP datagram is sent to a port where nothing is listening, Windows may raise WinError 10054 on recv calls. This function disables that behavior so the client does not crash if the emulator/server closes.

1.2 emulator module

emulator.py

UDP packet emulator to inject loss, corruption, duplication, and reordering. Works as a simple A<->Server relay.

1.2.1 Module overview

This tool sits between a client (A) and a real server, forwarding UDP datagrams in both directions while optionally introducing network impairments. It is used to test the robustness of a reliable data transfer protocol (e.g., selective retransmission, checksums, and in-order delivery), by simulating imperfect network conditions:

- Loss: drop selected packets
- Corruption: flip a random byte inside a packet
- Duplication: send the same packet twice
- Reordering: delay one packet briefly to reorder relative to others

`emulator.corrupt(data: bytes)` → bytes

Return a copy of 'data' with one randomly chosen byte bit-flipped (XOR 0xFF).

This simulates payload corruption on the wire.

Parameters
`data` – Original datagram bytes.

Returns
Corrupted datagram bytes (or unchanged if empty).

`emulator.emulator_loop(sock_a, addr_a, sock_b, addr_b, server_addr, args, stop_evt: Event)`

Main forwarding loop that applies impairments and relays packets between A and the server.

Parameters

- **sock_a** – UDP socket bound to the emulator's "A" listening address.
- **addr_a** – Tuple (ip, port) for the emulator "A" side (for logging).
- **sock_b** – UDP socket bound to the emulator's "B" listening address.
- **addr_b** – Tuple (ip, port) for the emulator "B" side (for logging).
- **server_addr** – Tuple (ip, port) of the actual server destination.
- **args** – Parsed argparse Namespace containing impairments probabilities.
- **stop_evt** – Event used to request graceful termination (e.g., on SIGINT).

Notes

- Sockets are set non-blocking and monitored with select.select().
- All send/recv operations are wrapped to prevent unexpected exceptions from exiting the loop; errors are logged and the loop continues.

`emulator.main()`

Parse CLI arguments, create/bind emulator sockets, and run the emulator loop.

Expected usage:

python emulator.py –listen-a 127.0.0.1:10000 –listen-b 127.0.0.1:10001
–server 127.0.0.1:12000 –loss 0.1 –corrupt 0.05 –reorder 0.05 –dup 0.02

Parameters

- **--listen-a** – IP:port where the emulator listens for the client ("A") side.
- **--listen-b** – IP:port where the emulator listens for the server-facing side.

- **--server** – IP:port of the actual server that the emulator forwards to.
- **--loss** – Probability [0..1] to drop a packet.
- **--corrupt** – Probability [0..1] to corrupt a packet.
- **--reorder** – Probability [0..1] to hold-and-delay (reorder) a packet.
- **--dup** – Probability [0..1] to duplicate a packet (send twice).

emulator.**win_udp_no_connreset**(*sock: socket*) → None

Disable Windows-specific UDP "connection reset" behavior for a socket.

On Windows, when a UDP datagram is sent to a port where nothing is listening, the OS may WinError 10054 on subsequent recv calls. This helper disables that behavior.

Parameters
sock – A bound UDP socket.

1.3 protocol module

protocol.py

This module defines the packet structure used by the Reliable Data Transfer (RDT) protocol. It provides:

- A fixed binary header format shared by both sender and receiver.
- Support for packing and unpacking packets into raw bytes suitable for UDP.
- A CRC32 checksum for data integrity validation.
- A compact payload limit chosen to enforce low data rate (< 500 bps) when combined with the enforced send delay in *rdt.py*.

class protocol.**Packet**(*seq_num: int*, *ack: bool*, *payload: bytes = b''*)

Bases: `object`

Represents a single RDT protocol packet, containing:

- Sequence number
- ACK flag (data or acknowledgement)
- Variable-length payload (0 to MAX_PAYLOAD bytes)
- CRC32 checksum for full data integrity

The class supports packing/unpacking into raw bytes suitable for UDP transmission.

pack() → bytes

Serialize the packet into raw bytes suitable for sending via UDP. CRC32 checksum is computed over the header (without checksum field) + payload.

Returns
Raw bytes that represent the packet on the wire.

static unpack(*buf: bytes*) → Tuple[*Packet*, bool]

Parse raw bytes into a Packet object and verify checksum integrity.

Parameters
buf – Raw UDP bytes received

Returns
(Packet object, checksum_valid flag)

Raises

`ValueError` – if buffer is too small to contain a header

1.4 rdt module

rdt.py

Selective Repeat reliable data transfer over UDP.

Key points: - Reliability: CRC32, per-packet timers, selective retransmissions - Reordering support with in-order delivery to the app - Very low send rate without congestion control:

> We enforce < 500 bps by using small payloads (32B) and a fixed 0.6s gap between packets.

1.4.1 Module overview

This module implements the sender/receiver state machines for a minimal Selective Repeat (SR) protocol over UDP. It relies on *protocol.Packet* for serialization and integrity (CRC32). The *RDTSession* class encapsulates all per-peer state:

- **Sender side:**
  - Sliding window (size = DEFAULT_WINDOW)
  - Per-packet timers and selective retransmissions
  - Throttled send rate (SEND_GAP) to satisfy the sub-500 bps requirement
- **Receiver side:**
  - Validates CRC32
  - Sends per-packet ACKs
  - Buffers out-of-order data; delivers in order to the application via a queue

`class rdt.`**`RDTSession`**(*sock: socket*, *peer: Tuple[str, int]*, *window: int = 8*, *timeout: float = 2.0*)

Bases: `object`

Per-peer Selective Repeat session.

`handle_raw`(*raw: bytes*)

Demultiplex a raw UDP datagram into ACK or DATA, verify checksum, and dispatch to the appropriate handler.

Parameters

`raw` – Raw datagram bytes as received from the socket.

`recv_available``()` → bytes

Retrieve any in-order bytes available for the application without blocking.

Returns

Concatenation of all currently queued in-order chunks (maybe empty).

Return type

bytes

`send`(*data: bytes*)

Reliable send with windowing and fixed inter-packet gap.

Splits ‘data’ into chunks of size at most MAX_PAYLOAD, then:

1) Waits until there is space in the sliding window.

2) Assigns a new sequence number and packs the chunk into a Packet.
3) Sends the packet to the peer and starts its retransmission timer.
4) Sleeps for SEND_GAP to keep the overall rate < 500 bps.

Parameters
`data` – Arbitrary bytes to send reliably to the peer.

`stop()`

Stop the retransmission thread and release resources owned by the session.

`rdt.now()` → float

Monotonic time helper used for timers (immune to wall-clock changes).

Returns
Current monotonic time in seconds.

Return type
float

1.5 server module

server.py

Single-thread UDP server with per-peer RDT sessions and a tiny file service.

The server exposes a minimal application-layer file transfer protocol on top of the custom reliable transport (RDT-Session). Each remote UDP address gets its own RDT session, allowing multiple independent clients to interact at once.

Supported commands (ASCII, newline terminated):

GET <filename>
-> server sends file contents back via RDT

PUT <filename>
-> client uploads raw file bytes immediately after

(any other text) -> server responds with “OK: ECHO: <text>”

`class server.PeerState`

Bases: `object`

Per-peer application state.

- inbuf: accumulates command-line bytes until ‘

‘ is seen

- mode: None or ‘receiving’ during PUT
- filename: target filename during PUT
- filebuf: bytearray of file content during PUT
- last_data_ts: last time we received any file bytes (for idle cutoff)

`on_file_bytes(data: bytes)` → None

`put_done()` → Tuple[str, int, bytes]

Finalize PUT state; return (filename, num_bytes, content).

`start_put`(*filename: str*) → None

`server.main()`

Main server loop.

Responsibilities:

- Bind a UDP socket and listen for incoming datagrams.
- Maintain a dictionary of per-peer RDTSession objects + app state.
- Feed raw packets into the proper session via handle_raw().
- Parse newline-terminated commands from a per-peer buffer.
- While in PUT mode, treat all bytes as file data until idle timeout.
- Shut down cleanly on Ctrl+C.

`server.win_udp_no_connreset`(*sock: socket*) → None

Windows-only: disable WSAECONNRESET behavior.

On Windows, if a UDP packet is sent to an unreachable port, the OS may raise WinError 10054 on subsequent recv calls. This helper disables that behavior so the server does not crash when a client exits abruptly.

PYTHON MODULE INDEX