PHASE 5 REPORT

EECE 4830
Network Design
TCP over Unreliable UDP
Dr. Vinod Vokkarane
Jacob Alicea · Josiah Concepcion · Jamie Oliphant · Tim Saari

Overview

This phase replaces the Phase 4 Go-Back-N transport with a compact Transmission Control Protocol that still uses the same lossy UDP socket. New pieces include a three-way handshake, reliable ordered delivery, time-outs driven by measured round-trip time, a sliding congestion window, and the usual Slow-Start plus additive-increase / multiplicative-decrease logic. The sender and receiver remain in Python, and the existing experiment script still sweeps loss rate, time-out value, and starting window size.

Design details

Each segment begins with a sixteen-bit Internet checksum and a thirty-two-bit sequence number, followed by up to 1024 bytes of data.

Connection setup follows the familiar SYN, SYN-ACK, ACK exchange; teardown finishes with a FIN.

Data are split into fixed blocks that carry ascending sequence numbers. The receiver accepts only the expected block and ignores anything out of order. A single timer guards the oldest unacknowledged block; when it expires every block still in flight is retransmitted, matching Tahoe behaviour.

Round-trip time is tracked with the exponential weighted moving average from RFC 6298. The retransmission time-out is always at least one millisecond and otherwise equals the estimated RTT plus four times the deviation.

Congestion control starts in Slow-Start, doubling the window every RTT until it reaches a threshold, then switches to additive-increase. A time-out halves the threshold and resets the window to one segment.

Key differences from Phase 4
• Handshake and FIN exchange replace the previous connection-less model.
• Reliability now covers a byte stream rather than a window of packets.

• Time-out adapts to measured RTT instead of using a fixed value.
• The congestion window grows automatically; the earlier fixed window is gone.

Running the code

Start the receiver with

```
python receiver.py --port 5003 --out_file received_tiger.jpg
--loss_rate 0.2
```

Start the sender in a second terminal with

```
python sender.py --ip 127.0.0.1 --port 5003 --file tiger.jpg
--loss_rate 0.2 --timeout 0.05 --init_window 1
```

To regenerate all performance graphs run

```
python experiment.py
```

followed by

```
python plot_phase5_perf.py
```

Testing

The same five scenarios used throughout the project were repeated: no loss, bit errors in acknowledgments, bit errors in data, lost acknowledgments, and lost data. Each point is an average of three transfers of a 650 kB JPEG.

Performance highlights

Figure 1 shows RTT rising quickly during Slow-Start and settling near twenty milliseconds once congestion control stabilises.
Figure 2 shows the time-out following RTT with the required safety margin.
Figure 3 shows the congestion window reaching the threshold in under half a second and then growing linearly.
Figure 4 indicates completion time is flat up to forty percent random loss and grows sharply beyond that.
Figure 5 reveals that time-outs between ten and one-hundred milliseconds perform similarly; fifty milliseconds is slightly worse because of an unlucky interaction with RTT.
Figure 6 demonstrates that once Slow-Start begins, the initial window size matters only for the first few round-trips.

Observations

- Tahoe’s loss recovery copes with up to forty percent random drops before the window collapses.
- Choosing a slightly long time-out is safer than choosing one that is too short.
- After a handful of RTTs, the starting window no longer influences throughput.
- Deviation in RTT shrinks once the queue stabilises, confirming congestion control is effective.

Conclusion

Adding true TCP semantics required roughly one hundred eighty extra lines of code yet produced a far more resilient protocol. Throughput now scales automatically, only the missing segment is resent after a loss, and adaptive time-outs avoid needless retransmissions on longer paths. On networks with up to forty percent random loss the 650 kB file now transfers in roughly six to seven-tenths of a second, about ten times faster than the earlier Go-Back-N design. Future improvements could include Reno fast-retransmit or delay-based tuning; duplicate-ack counting and base RTT measurement are already in place.

Appendix

main files
sender.py and receiver.py: core TCP logic with handshake, checksum, congestion control, and RTT logging

utils.py: Internet checksum helper

experiment.py: batch driver for loss, time-out, and window sweeps

Plot_phase5_perf.py: creates the performance figures
phase5_perf.csv and tcp_timeseries.csv: raw measurements
PNG images