Course	COMP 7003
Program	Bachelor of Science in Applied Computer Science
Term	September 2025

- This is an individual tools assignment.

Objective

- Deepen your understanding of transport-layer behaviour by generating controlled TCP and UDP traffic, capturing it, and analyzing how each protocol behaves under clean, lossy, and bottlenecked network conditions. You will use iperf3, tcpdump, Wireshark, and tc/netem to run experiments, generate evidence, and justify which protocol is more suitable for different types of real-time communication.

Learning Outcomes

- Generate controlled TCP and UDP flows with `iperf3`.
- Capture and analyze packet-level behaviour using `tcpdump` and Wireshark.
- Use `tc` with `netem` to introduce delay, loss, and bandwidth limits.
- Identify slow start, congestion avoidance, retransmissions, head-of-line blocking, loss, and jitter.
- Connect measured transport behaviour to real-time application requirements.

Assignment Details

- You will compare TCP and UDP across three controlled experiments. All testing must be performed between two Linux hosts:
- Host Role Tools Used
- Host A `iperf3` client, `tcpdump` sender-side capture `iperf3`
- Host B `iperf3` server, `tcpdump` receiver-side capture `iperf3`
- Either host may run `tc/netem` to introduce delay, loss, or bottlenecking.
- Your test parameters must remain consistent across comparable runs so that the protocol and path differences produce the observed behaviour.
- Remember to remove all existing `tc/netem` rules at the end of your session to prevent causing issues for other users of the hosts.

Experiment Conditions

- You will run three conditions.
- For each condition, you do two iperf3 tests: one TCP run and one UDP run.
- Every run must be captured with tcpdump and analyzed in Wireshark.
- Ultimately, you should have a total of 6 tests: 3 conditions × 2 protocols.

Step 1: Clean Path (Baseline)

- First, measure TCP and UDP on a normal link with no artificial problems.
  - Remove all `tc/netem` rules so the path has no added delay and no added loss.
  - Run one TCP test with `iperf3` (for example, 20 seconds from Host A to Host B).
  - Run one UDP test with `iperf3` using a fixed, moderate bitrate (for example, 5 Mbit/s for 20 seconds).
  - Capture both runs with `tcpdump` and analyze them in Wireshark to see how quickly TCP ramps up compared to UDP's immediate fixed sending rate.
- This step gives you the baseline startup behaviour for both protocols.

Step 2: Moderate Random Loss

- Next, add controlled loss and see how each protocol reacts.
  - Use `tc/netem` to add about 1% random packet loss on one interface in the path.
  - Repeat the same TCP test as in Step 1 (same direction, duration, and basic settings).
  - Repeat the same UDP test as in Step 1 (same bitrate and duration).
  - Capture both runs with `tcpdump` and analyze them in Wireshark to observe TCP retransmissions, throughput drops and recovery, and UDP's constant send rate with increased loss and jitter.
- This step illustrates how TCP prioritizes throughput over reliability, while UDP maintains timing but discards data.

Step 3: Bottlenecked Link (Overload)

- Finally, create a bottleneck and push UDP past it.
  - Remove all existing `tc/netem` rules so the path has no added delay and no added loss.
  - Use `tc` (for example, `netem` combined with a rate-limiting `qdisc`, such as tbf or htb) to set a bandwidth cap on one interface, starting with a value of 10 Mbit/s.
  - Keep the bandwidth cap fixed and increase your UDP send rate until the receiver shows clear packet loss.
  - Run a TCP test with `iperf3` using default settings (for example, 20 seconds). TCP will try to fill the link, but should adapt to the limit.

    - Run a UDP test with `iperf3` at a bitrate higher than the limit (for example, 20 Mbit/s on a 10 Mbit/s link) for the same duration.
    - Capture both runs with tcpdump and analyze them in Wireshark to compare TCP converging near the bottleneck rate versus UDP continuing at the configured rate, which causes high loss and jitter.
- This step shows TCP's self-throttling behaviour compared to UDP's tendency to overload the link if the application does not manage its rate.

Report

- Your report must have two sections.

Results for the Three Conditions

- Make three subsections:
    - Clean path
    - Moderate loss
    - Bottlenecked link

- For each subsection, do the same thing.
- Split the subsection into TCP and UDP.

TCP

1. Show the test parameters:
    a. Include the iperf3 command you used and any tc/netem settings active during the test.

2. Show the relevant Wireshark data.
    a. Include the graph or view that best demonstrates TCP behaviour for this condition (for example, startup curve, sawtooth behaviour, convergence near the cap).

3. Explain what you see.
    a. Write one short paragraph describing the key behaviour you observed, such as ramp-up, throughput stability, loss recovery, or adaptation to the bottleneck.

UDP

1. Show the test parameters:
    a. Include the iperf3 UDP command (bitrate, duration) and the tc/netem settings for this condition.

2. Show the relevant Wireshark data:

   a. Include the graph or view that shows the UDP behaviour you care about (for example, immediate fixed rate, loss gaps, jitter, receive rate vs send rate).

3. Explain what you see:
   a. Write one short paragraph describing the key behaviour, such as constant sending rate, packet loss, jitter, or how the link cap affected the received rate.

Discussion and Conclusion

- Use your own results to answer the “so what” questions.
- In this section:
  - Explain how TCP’s behaviour (slow start, congestion control, retransmissions, head-of-line blocking when loss occurs) helps or hurts fast movement updates in a real-time game.

  - Explain how UDP’s behaviour (low latency, constant send rate, no built-in reliability or backoff) helps or hurts the same kind of updates.

  - For a Cyber Tag style game, state which protocol you would use for:
    - frequent movement and position updates,
    - rare but critical events (for example, “player tagged” state changes),
    - non-real-time traffic such as login, lobby, or chat.

  - Justify each choice with a sentence or two that points back to what you saw in your graphs and captures, not to generic theory.
  - Finish with a conclusion that summarizes your main findings.

Constraints

- Use Linux on both hosts.
- Use `iperf3`, `tcpdump`, Wireshark, and `tc/netem`.
- Use your own commands and your own graphs.
- Do not copy other students’ commands, captures, or screenshots.
- Keep all tests within the lab or home network.

Resources

- Provided notes

Submission

- Ensure your submission meets all the guidelines, including formatting, file type, and submission.
- Follow the AI usage guidelines.
- Be aware of the late submission policy to avoid losing marks.
- ***Note: Please strictly adhere to the submission requirements to ensure you don't lose any marks.***

Evaluation

Topic	Value	Details
Data collection and capture	30%	All six tests (3 conditions × TCP/UDP) completed; tcpdump captures valid; parameters and conditions clearly recorded.
Results: TCP/UDP sections	30%	For each condition, TCP and UDP subsections include test parameters, at least one (preferably more) appropriate Wireshark graph or view, and a clear paragraph explaining what is observed.
Transport behaviour analysis	30%	Correct interpretation of TCP startup, congestion behaviour, and loss handling; correct interpretation of UDP rate, loss, and jitter; at least one clear TCP loss example and one clear UDP loss/jitter example.
Discussion and conclusion	10%	Coherent, evidence-based argument about when to use TCP vs UDP for different traffic types in a real-time game; clear final recommendations tied directly to the measured results.
Total	100%	

Hints

- Run short test commands first to ensure that paths, ports, and captures work correctly before running the complete tests.
- Name your pcap files clearly, for example: clean-tcp.pcap, loss-udp.pcap, bottleneck-tcp.pcap.
- Keep your `iperf3` parameters identical across conditions except for the condition itself (loss or bottleneck).

- Use `tc qdisc show` before and after each condition to confirm precisely what is active.
- In Wireshark, the TCP Time-Sequence Graph (tcptrace) is the easiest way to visualize slow start, congestion avoidance, and recovery after loss.
- Wireshark’s I/O Graph helps show both the send and receive rates as two lines over time.
- Run tcpdump with a filter to keep your files small.