COMP3014 Network Simulation Project
Part C: TCP Flavours

Nima Afraz and Abdul Wadud

October 15, 2025

Introduction

Transmission Control Protocol (TCP) controls how much data it transmits over a network by utilising a sender-side congestion window and a receiver side advertised window. TCP cannot send more data than the congestion window allows, and it cannot receive more data than the advertised window allows. The size of the congestion window depends upon the instantaneous congestion conditions in the network. When the network experiences heavy traffic conditions, the congestion window is small. When the network is lightly loaded the congestion window becomes larger. How and when the congestion window is adjusted depends on the form of congestion control that the TCP protocol uses. Congestion control algorithms rely on various indicators to determine the congestion state of the network. For example, packet loss is an implicit indication that the network is overloaded and that the routers are dropping packets due to limited buffer space. Routers can set flags in a packet header to inform the receiving host that congestion is about to occur. The receiving host can then explicitly inform the sending host to reduce its sending rate. Other congestion control methods include measuring packet round trip times (RTTs) and packet queuing delays Some congestion control mechanisms allow for unfair usage of network bandwidth, while other congestion control mechanisms are able to share bandwidth equally [EA12]. In this project we will simulate four TCP algorithms to evaluate their performance under similar conditions: TCP Cubuc, TCP Reno, TCP Yeah, and TCP Vegas (more information on these algorithms can be found in [MBR13]).

1 TCP Congestion Control Algorithm Simulation

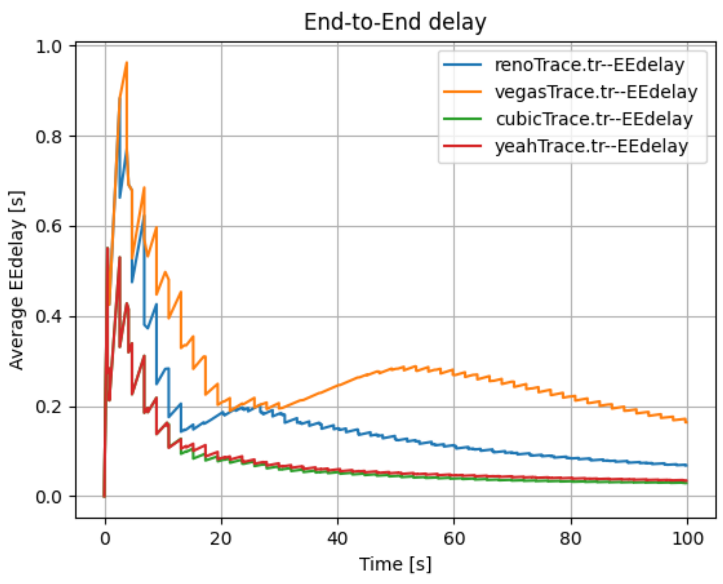
Run the following 4 scenarios:

```bash
#! /bin/bash
cd comp3014j
ns cubicCode.tcl ; ns yeahCode.tcl ; ns renoCode.tcl ; ns vegasCode.tcl
```

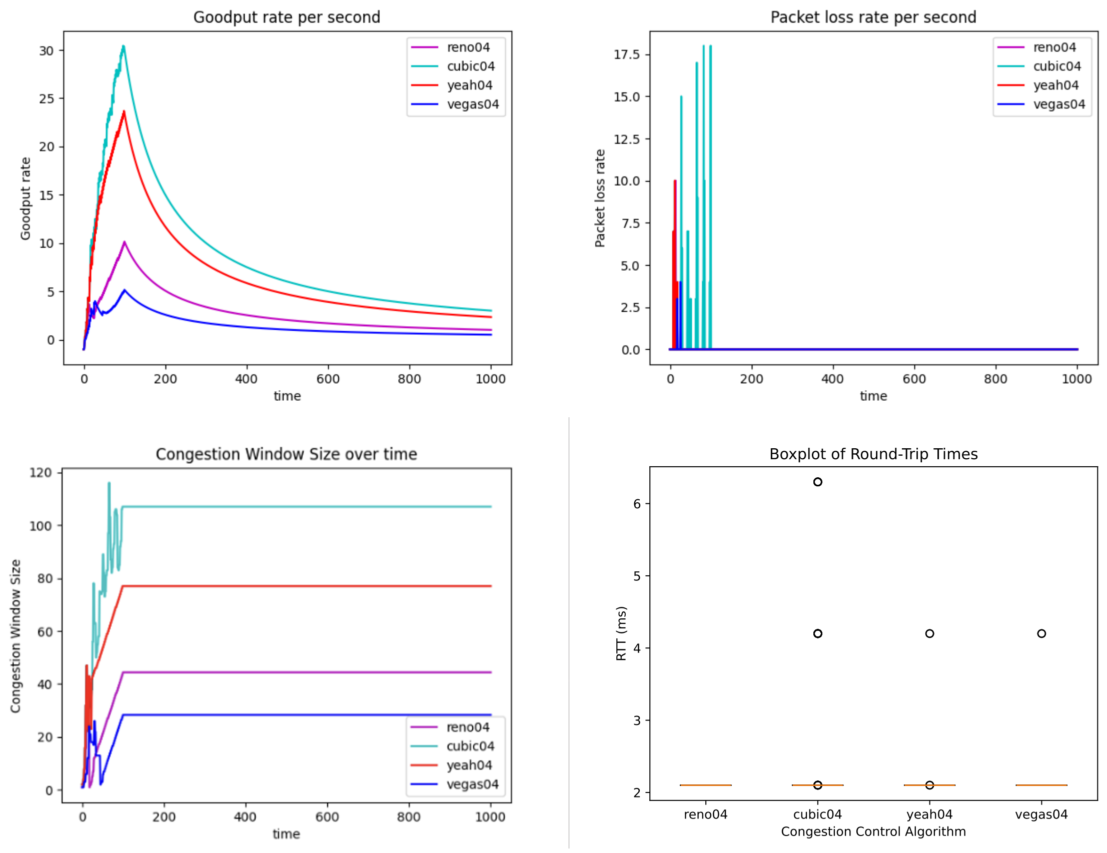
The above scenarios have exactly the same topology with the only difference that they use different TCP congestion control algorithm. We expect to observe different results (packet loss and Goodput) for each algorithm.

The above command will run the 4 scenarios and will generate trace (.tr) and nam files for each scenarios. The next line will use the traceanalyser package to parse and plot the end-to-end delay of the 4 TCP algorithm.

```bash
#! /bin/bash
python3 analyser2.py
```



```bash
#! /bin/bash
python3 analyser.py
```



2 You will write the analyser (Python)

You must write a new Python script `analyser3.py` **from scratch** (following the templates provided). Your new analyser script must answer the questions of Part A.

2.1 Part A (40 points): TCP Variants, Fixed Topology

Run the four variants with the default topology and traffic.

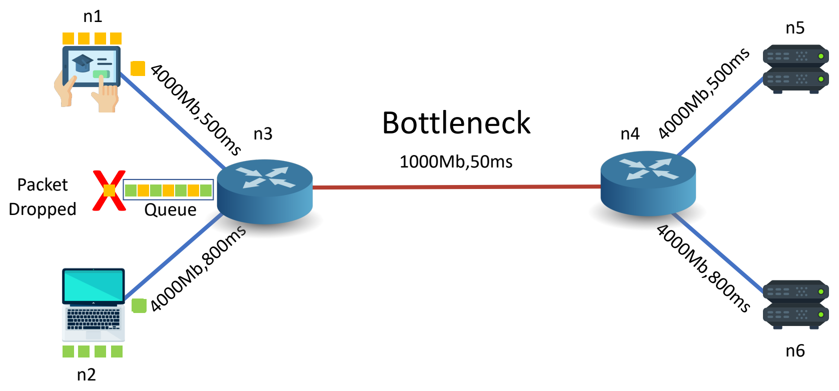
1. (10) Table: per-flow `total_goodput_Mbps` and `plr_pct`. Add **one** comparison figure (two subplots: goodput, PLR).
2. (10) Compute **Jain's fairness index** over the last third. Which variant is fairest? One-paragraph explanation.
3. (10) **Throughput stability:** which variant is most stable (lowest CoV)? One paragraph relating to its growth/backoff logic.
4. (10) **Short conclusion:** Which algorithm is "best" under this setup and why? (3–5 sentences.)

Hint: Jain's fairness index, you can use the following formula:

$$J(x_1, x_2, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

where x_i is the throughput (or resource share) of flow i, and n is the total number of flows. The index ranges from 0 (worst fairness) to 1 (perfect fairness).

3 Queuing Algorithms



In computer networks, each router has a queue where it stores packets if the link ahead of it is congested. These queue are finite meaning that they have a limitation on the number of packets they can store. After reaching this limit the queue should drop some of the packets to make space for newer or more important packets. In this part of the project, we will learn about different algorithms used in networks to decide which packets to drop in a queue:

1. **Drop Tail:** Drop Tail is a simple queue management algorithm used by network schedulers in network equipment to decide when to drop packets. With drop Tail, when the queue is filled to its maximum capacity, the newly arriving packets are dropped until the queue has enough room to accept incoming traffic. Its name arises from the effect of the policy on incoming packets. Once a queue has been filled, the router begins discarding all additional datagrams, thus dropping the tail of the sequence of packets. The loss of packets causes the TCP sender to enter a slow-start, which reduces throughput in that TCP session until the sender begins to receive acknowledgements again and increases its congestion window.
2. **Random Early Detection (RED):** RED is a type of congestion control algorithm/mechanism that takes advantage of TCP's congestion control mechanisms and takes proactive approach to congestion. RED drops packets with non-zero drop probability after the average queue size

exceeds a certain minimum threshold, it doesn't wait for the queue to be completely filled up. Its objectives are to provide high link utilization, remove biases against busty sources, attenuate packet loss and queuing delay, and reduce the need of global synchronization of sources. RED was proposed to reduce limitations in Drop Tail, it is an active queue technique (more about queuing in [OEK18]).

3.1 Simulation different queuing algorithms

In this section we will compare two queuing algorithms available in ns2 with the same topology and scenario to compare the results.

Change the queuing algorithm of the bottlneck link from DropTail to RED in the .tcl files (line 30 of the code in "cubicCode.tcl, yeahCode.tcl, renoCode.tcl, vegasCode.tcl")

```
# Change DropTail to RED in the Bottleneck link.

#From:
$ns duplex-link $n3 $n4 1000Mb 50ms DropTail

#To:
$ns duplex-link $n3 $n4 1000Mb 50ms RED
```

After changing the queuing algorithm of the bottleneck link now you need to run the simulations again to generate new trace files.

```
#! /bin/bash
cd comp3014j
ns cubicCode.tcl ; ns yeahCode.tcl ; ns renoCode.tcl ; ns vegasCode.tcl
```

Now run both analyser and analyser2 scripts to generate the new results.

```
#! /bin/bash
python3 analyser3.py     # your newly built analyser
python3 analyser2.py
python3 analyser.py
```

3.2 Part B (35 points): DropTail vs RED (Minimal Workload)

Scenarios: Run *once* with DropTail and *once* with RED (same topology). Optionally pick one smaller capacity (e.g., 500 Mb/s) *or* one larger (e.g., 2 Gb/s) for a quick sensitivity check (**choose only one** extra; total 3 runs).

1. (15) Compare **goodput, PLR, fairness, stability(CoV)** between DropTail and RED using **one** figure (two subplots max). One paragraph of interpretation.

2. (20) Sensitivity (single added capacity): Does congestion level change which queue is preferable? One figure + 150–250 words.

3.3 Part C (25 points): Light Reproducibility

1. (15) Pick **one** scenario (your most interesting) and repeat **5 runs** with different seeds or start-time jitters. Show mean and 95

2. (10) **Packaging:** a single shell script that (i) runs sims, (ii) runs the analyser, (iii) regenerates CSVs and plots.

References

[EA12] Annali Esterhuizen and Krzesinski Aes. Tcp congestion control comparison. 2012.

[MBR13] Luis Armando Marrone, Andrés Barbieri, and Matías Robles. Tcp performance – cubic, vegas amp; reno. *Journal of Computer Science and Technology*, 13(01):p. 1–8, Apr. 2013.

[OEK18] N. Obinna Eva and L. G. Kabari. Comparative Analysis of Drop Tail, Red and NLRED Congestion Control Algorithms Using NS2 Simulator. *International Journal of Scientific and Research Publications (IJSRP)*, 8(8), August 2018.