Project 12: Network Congestion Control Simulator

Gaurav Budhwani (22110085) Kaveri Visavadiya (22110114)

Monday 15th December, 2025

Instructor: Sameer Kulkarni

Contents

1 Objective

This report presents the progress of our project, which aims to create an interactive simulation platform that demonstrates how different TCP congestion control algorithms (Reno, CUBIC, BBR) work under varying network conditions, such as bandwidth, router buffer size, delay and topology of the network. The goal is to provide real-time visualization of congestion window (cwnd), throughput, inflight packets and packet loss with time.

The project is divided into two components: a frontend and a backend. The backend consists of ns3, which runs the network simulation and generates data, and Python, which processes the data and controls the simulation via Python bindings. The frontend consists of React/Javascript that communicates with the Python backend to visualize data.

2 Introduction and Theoretical Background

The goal of this project is to create an education tool to explain congestion control for networking students and instructors. The application should be intuitive to use and should explain these concepts simply, displaying the relevant graphs in real time.

Out of the 7 layers of the OSI model, our work lies in the application, transport and network layers. At the **application layer**, the Flask backend acts as the interface between the user and the simulator, receiving inputs from the frontend and returning simulation results. The **transport layer** is where most of the simulation logic happens — this is where the TCP variants Reno, Cubic, and BBR are implemented and tested, handling congestion control, acknowledgments, and window management. At the **network layer**, ns-3 manages IPv4 addressing, routing, and packet forwarding between nodes, while our Python routing utilities help determine the paths data packets take through the simulated topology.

In this introduction, we will look at the Transmission Control Protocol and how it mitigates congestion in the network with the help of various congestion control algorithms like Reno, CUBIC and BBR.

2.1 Introduction to TCP Congestion Control

Services provided by TCP: TCP (Transmission Control Protocol) provides

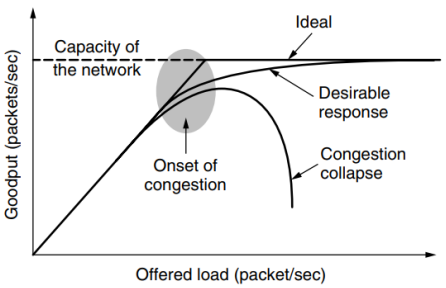
1. reliable end-to-end byte stream over an unreliable internetwork between hosts.
2. handling for retransmissions and timeouts.
3. communication service at an intermediate level between the application layer and the network layer.

The sending and receiving TCP hosts exchange data in the form of segments. A segment consists of a fixed 20-byte header (plus an optional part) followed by 0 or more data bytes, and its size is constrained by the MTU (Maximum Transfer Unit) of each link (which is ∼1500 bytes, the Ethernet payload size). Every data byte on a TCP connection has a 32-bit sequence number which is used to number each byte of data transmitted.

TCP uses a sliding window protocol with a dynamic window size to transmit data with a self-clocking mechanism. When the sender transmits a segment, it also starts a timer on its end. When this segment arrives at the destination, the receiver transmits a segment with an acknowledgment (ACK) number equal to the next sequence number it expects to receive and the remaining window size. This window size determines the number of bytes that can be sent by the sender for the next RTT. If the sender's timer goes off before the ACK is received, it retransmits the segment again. TCP is responsible for retransmission of lost data, reassembling out-of-order data and minimizing network congestion. Once the TCP receiver has reassembled the sequence of octets originally transmitted, it passes them to the receiving application. Thus, TCP (or the transport layer as a whole) abstracts away the application's communication from the underlying networking details.

Congestion occurs when the offered load exceeds what the network (links, routers) can forward and packets pile up in queues faster than they can be transmitted, causing packet loss or long queueing delays. Increasing load beyond a certain point can cause packets to be retransmitted excessively due to losses and delays, leading to congestion collapse where goodput drops sharply.

Figure 1: Congestion in the network



The congestion window (cwnd) is a sender-side variable in TCP that limits the amount of unacknowledged data a sender can transmit into the network at any given time. The actual amount of data the sender is allowed to have "in flight" (unacknowledged) is the minimum of the congestion window and the receiver's advertised window size: Transmit Window Size $= min(cwnd, rwnd)$. Morevoer, it is unrelated to the receiver window (rwnd), which is limited by the receiver's buffer space; the congestion window is an internal calculation performed by the sender's OS and is not communicated across the network.

AIMD Control Law for Congestion Control: In the absence of a congestion signal (either packet loss or ECE), senders should increase their rates, and when a congestion signal is given, senders should decrease their rates. Since it is easy to drive the network into congestion and difficult to recover, the increase policy should be gentle and the decrease policy should be aggressive. TCP uses the AIMD (additive increase multiplicative decrease) control law to arrive at an efficient and fair operating point that allocates bandwidth equally between users. Instead of adjusting the sending rate directly, it adjusts the size of the sliding congestion window (cwnd).

2.2 TCP Congestion Control Algorithms

2.2.1 TCP Reno

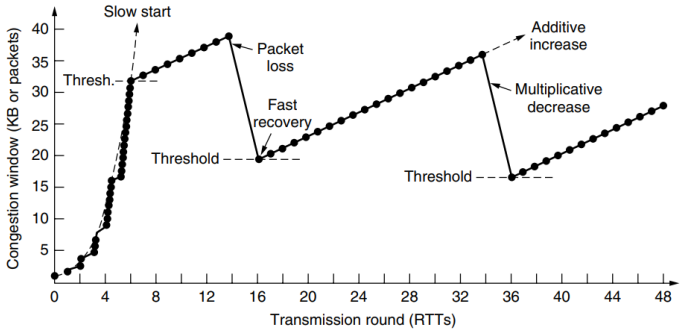


Figure 2: Fast recovery and sawtooth pattern of TCP Reno

1. **Slow start:** At the beginning of a connection, the cwnd starts at a small value (atmost 4 MSS) to probe the network for bandwidth. cwnd effectively doubles every RTT.
2. **Congestion Avoidance:** When the cwnd reaches the slow start threshold, its growth changes from exponential to linear (additive increase). The sender increases the cwnd by one MSS every RTT until loss detection occurs.
3. **Congestion Detection & Reduction:** TCP detects congestion through implicit signals like packet loss (indicated by retransmission timeouts or the receipt of duplicate acknowledgments or ECE bit set in the receiver's message). Duplicate ACKs signals to the sender, without waiting for a potentially long retransmission timeout (RTO), that a segment might have been lost or delivered out of order. If the sender receives 3 duplicate ACKs in a row, it is considered a strong indication that the corresponding segment is lost. This triggers the **Fast Retransmit** mechanism, where the sender immediately retransmits the missing segment without waiting for the retransmission timer to expire.
    - If a timeout occurs, the cwnd is typically reset to 1 MSS, and the process restarts with the slow start phase.
    - If duplicate ACKs are received, the cwnd is usually halved (multiplicative decrease), and the sender enters the **Fast Recovery** state, a less drastic response than a timeout.

2.2.2 TCP CUBIC

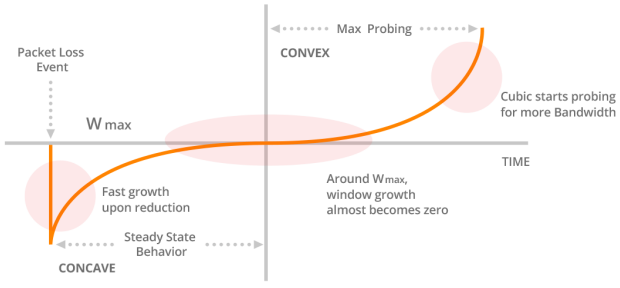


Figure 3: Evolution of TCP CUBIC Congestion Window

Some of the features of TCP CUBIC are:

1. **RTT Independence:** CUBIC's window growth is time-dependent rather than RTT-dependent. This provides better fairness among flows with different RTTs, as a flow with a longer RTT can still grow its window aggressively, unlike in TCP Reno where flows with shorter RTTs grow faster.
2. **Cubic Growth Function:** After a packet loss, CUBIC reduces its window multiplicatively (by 30%) and then enters a **concave** growth phase, quickly returning to near its previous maximum window size (W_{max}).

- As it approaches the previous W_{max}, its growth rate slows down to probe the network for available capacity.
- If no loss is detected beyond the previous W_{max}, it enters a **convex** growth phase, where the window size starts increasing rapidly again to explore new available bandwidth.

3. **Suitable in environments with high bandwidth-delay product:** Networks with large bandwidth-delay product take many RTTs to reach the available capacity of the link. CUBIC's aggressive, non-linear growth allows it to utilize high-bandwidth links more effectively and reach link capacity faster than linear algorithms like TCP Reno.

2.2.3 TCP BBR

TCP BBR (Bottleneck Bandwidth and Round-trip time) is a modern congestion control algorithm developed by Google that continuously estimates the available bandwidth and round-trip time (RTT), instead of relying on packet loss. BBR can achieve high throughput and low latency in high-speed or lossy networks by avoiding the aggressive slowdowns caused by packet loss signals from traditional methods. Importantly, it avoids bufferbloat, which is the buildup of large data buffers that can significantly increase latency.

3 Backend Python Implementation of Congestion Control Algorithms

The function **run_simulation(algorithm, bw_mbps, delay_ms, buffer_size, duration, mss_bytes)** in the file backend/app.py is a deterministic, discrete-packet, time-stepped computational model that emulates the fundamental feedback dynamics of TCP congestion control. The simulation advances in fixed time increments (Δt, 0.01s), with all traffic-related quantities—such as `buffer_current`, `inflight`, and `dropped`—represented as integer-valued packets rather than continuous flow variables.

Each simulation cycle consists of four sequential stages: (1) **Packet Transmission**, (2) **Queueing and Dropping**, (3) **Buffer Draining**, and (4) **Acknowledgment Generation**. The sender's transmission rate is governed by the current congestion window (*cwnd*) and the base round-trip time (RTT_{base}), yielding a discrete packet credit that determines the number of packets injected into the network buffer. When the queue occupancy exceeds its capacity (B_{max}), surplus packets are recorded as *dropped*, while the buffer drains at a rate defined by the link capacity (C_{link}) expressed in packets per second. Drained packets are subsequently scheduled for acknowledgment.

3.1 Feedback Dynamics and Acknowledgment Scheduling

The *feedback control mechanism* is implemented through an acknowledgment scheduling structure (`ack_schedule`), which models the temporal delay between successful transmission and acknowledgment reception. Each drained packet is assigned a round-trip time sample:

$$RTT_{sample} = RTT_{base} + D_{queue}$$

where D_{queue} represents the instantaneous queueing delay. At each time step, acknowledgments that have reached their scheduled delay are retrieved (`acked = ack_schedule.pop(0)`), providing integer-valued feedback to the TCP algorithm, alongside the number of packets lost.

The TCP algorithm, representing variants such as TCP Reno, Cubic, and BBR, processes these feedback signals to update its internal state variables—primarily the congestion window (*cwnd*) and the slow-start threshold (*ssthresh*)—according to its characteristic control law (e.g., additive increase/multiplicative decrease, cubic growth function, or bandwidth-probing mechanism).

3.2 System Architecture and Communication Interface

The system adopts a *client–server architecture* integrating a React.js frontend with a Python Flask backend. The frontend acts as the control and visualization interface, while the backend functions as the computational core. Simulation parameters—including the congestion control algorithm, link bandwidth, propagation delay, and buffer capacity—are serialized into a JSON object and transmitted via HTTP POST (`axios.post`) to the Flask endpoint `/simulate`. Upon receipt, the backend executes the `run_simulation()` routine with the supplied configuration, generating a structured time-series output (`trace`). This dataset is serialized as a JSON response and returned to the frontend, where it updates the application state (`setData(res.data.trace)`), triggering real-time visualization of throughput, congestion window evolution, and packet loss.

3.3 Sender–Receiver Abstraction

Both the sender and receiver are modeled as logical abstractions rather than full network stacks. The **Sender** encapsulates the TCP algorithmic logic, responding to the feedback signals (`acked` and `dropped`) to adapt the congestion window and

transmission rate in accordance with its control policy. The **Receiver** is implicitly represented through the acknowledgment scheduler: each successfully drained packet generates an acknowledgment after its corresponding RTT_{sample} delay, emulating bidirectional feedback propagation without explicitly modeling receiver-side logic.

4 Backend ns3 Usage to Simulate Congestion Control Algorithms

ns-3 is an open-source, discrete-event network simulator used for research, teaching, and protocol development. It models packet-level behavior of networks (TCP/UDP, IP, routing, Wi-Fi, LTE, etc.) with realistic link parameters (bandwidth, delay, queues) and traffic/application helpers. ns-3 is powerful for comparing transport algorithms (e.g., Reno/Cubic/BBR), studying bufferbloat, testing routing strategies, and validating ideas before real-world deployment.

`run_simulation_ns3` in `app.py` and the program `tcp_multi.cc` work together as a pipeline that runs a full packet-level ns-3 experiment and converts its sampled TCP metrics into the JSON trace that the frontend visualizer uses. The logic is straightforward: `run_simulation_ns3` builds a command line from the UI parameters (algorithm, rate, delay, buffer, duration, MSS) and launches ns-3 as a subprocess in the `ns3` directory with the appropriate arguments. After ns-3 finishes, it checks that the CSV output was created, opens `trace_flow0.csv`, parses each row, and converts the columns (`time`, `cwnd_pkts`, `throughput_mbps`, `buffer_pkts`, `inflight_pkts`) into a Python list of dictionaries and returns that list to the `/simulate` route as JSON.

`tcp_multi.cc` is the ns-3 experiment that produces those CSV files. On startup, it parses the same arguments (`--flows`, `--rate`, `--delay`, `--bufferPkts`, `--duration`, `--mss`, `--sampleDt`), sets global TCP parameters (MSS, socket buffers), constructs a simple topology (N senders → single router → N receivers), configures the upstream and bottleneck links (fast sender→router links, configurable router→receiver bottleneck with the `rate` and `delay` arguments), and installs a queue discipline (FIFO, CoDel, etc.) on the bottleneck interface sized in packets. For each configured flow, the program chooses a TCP `TypeId` (Reno/Cubic/BBR), installs a `BulkSend` sender app and `PacketSink` receiver app, defers trace hookups until sockets exist, and then periodically samples metrics via `DoSample`. The sampler reads per-flow `CongestionWindow` and `BytesInFlight` (via ns-3 `Config` trace hooks) and counts received bytes to compute throughput; it also reads the router qdisc packet count to report queue occupancy. Each sample is written as one line in `trace_flow{i}.csv` with the columns `time, cwnd_pkts, throughput_mbps, buffer_pkts, inflight_pkts`.

Together these pieces enable the frontend visualizer to show real-time plots of congestion-control behavior.

5 Frontend Architecture and Functionality Analysis

The frontend is a React-based single-page application that provides an interactive visualization interface for TCP congestion control simulations. It consists of two main components that handle single-flow and multi-flow simulation modes, communicating with the Flask backend via REST API calls.

5.1 App.jsx

`App.jsx` is the top-level controller and visualizer for the TCP congestion-control application. It keeps the global UI state (engine selection, single vs. multi mode, simulation parameters, playback time and speed, loading and error flags) and handles all communication with the backend via HTTP requests. When the user starts a run the component posts the current configuration to the backend endpoints (single-run or CSV download), receives JSON traces, normalizes them into a predictable shape, and drives a simple playback loop that advances a time cursor. The component filters trace samples up to the current time and supplies those samples to the charting components (CWND, throughput, inflight, queue/loss) and the small sender–router–receiver schematic. `App.jsx` also adapts its display depending on the data source (Python model vs ns-3), supports CSV export, surfaces backend debug information, and in multi-flow mode delegates topology editing to `TopologyBuilder` while rendering per-flow and per-link charts returned by the backend.

5.2 TopologyBuilder.jsx

`TopologyBuilder.jsx` manages topology selection (single, parallel, series, triangle, branched, mesh), maintains lists of senders and receivers (with router attach points), and allows the user to create flows (source, destination, algorithm). It exposes global per-link defaults (bandwidth, delay, buffer, duration, sampling interval) and supports per-link overrides. The component validates node names and attachments, computes the logical link set for the chosen topology, renders a simple SVG diagram of nodes and links, and offers a link editor UI for fine-grained parameter edits. When the user clicks "Run Multi" TopologyBuilder builds a clean payload (topology, senders, receivers, flows, link parameters and overrides), POSTs it to the backend `/simulate_multi` endpoint, and then forwards the returned traces and debug data to the parent via callbacks so the parent can start playback and render per-flow and per-link visualizations.

Together, `App.jsx` and `TopologyBuilder.jsx` separate concerns: `TopologyBuilder` composes multi-flow experiments and sends the run request, while `App.jsx` handles single-run control, overall playback, charting and presentation of results. The frontend posts the user-configured parameters to the backend, receives time-series traces (single or multi), and animates those traces so the user can visually compare congestion-control algorithms and per-link behavior.

6 Results and Other Considerations

6.1 Single Flow Topology

Figure 4: TCP (Python) results

For **TCP Reno**,

- cwnd vs time graph is as expected.
- Throughput ($throughput = min(\frac{cwnd}{throughput}, bandwidth)$ drops drastically at the time instants when cwnd halves. At the rest of the time instants, the minimum cwnd (which seems to be ∼30 Mb) / RTT (=100ms) is still larger than the bandwidth of 5 Mbps, so the throughput remains largely constant.
- Inflight packets (which are always less than equal to the cwnd size) roughly follow the same trajectory as the cwnd graph. It is not the exact same as the cwnd graph because the congestion window (cwnd) graph shows the limit the sender imposes on itself to prevent network congestion, while the inflight packets graph shows the actual amount of data currently unacknowledged in the network.
- Since Reno is a loss-ful TCP congestion control mechanism, i.e., the sender reduces its datarate upon receiving packet loss notification, the graph of packet loss vs time increases stepwise at the time instants when cwnd halves.

For **TCP CUBIC**,

- The graph of cwnd vs time is, as expected, a roughly cubic function of time.
- Similarly, the graphs of throughput, inflight packets and packet loss can be similarly argued for their correctness.

For **TCP BBR**,

- The graph of cwnd vs time is constant over time as because BBR doesn't grow the congestion window based on packet loss like Reno or Cubic. Instead, it estimates the bottleneck bandwidth and round-trip time (RTT) and keeps CWND just large enough (= 2 × BDP) to fill the pipe without building queues. Since those estimates remain stable in steady conditions, the congestion window stays roughly constant over time.
- The throughput graph shows small oscillations due to BBR's bandwidth probing behavior, while the cwnd graph remains constant since it's tied to a stable bandwidth-delay product estimate.
- Since BBR is a lossless algorithm, the graph of packet loss vs time is 0.

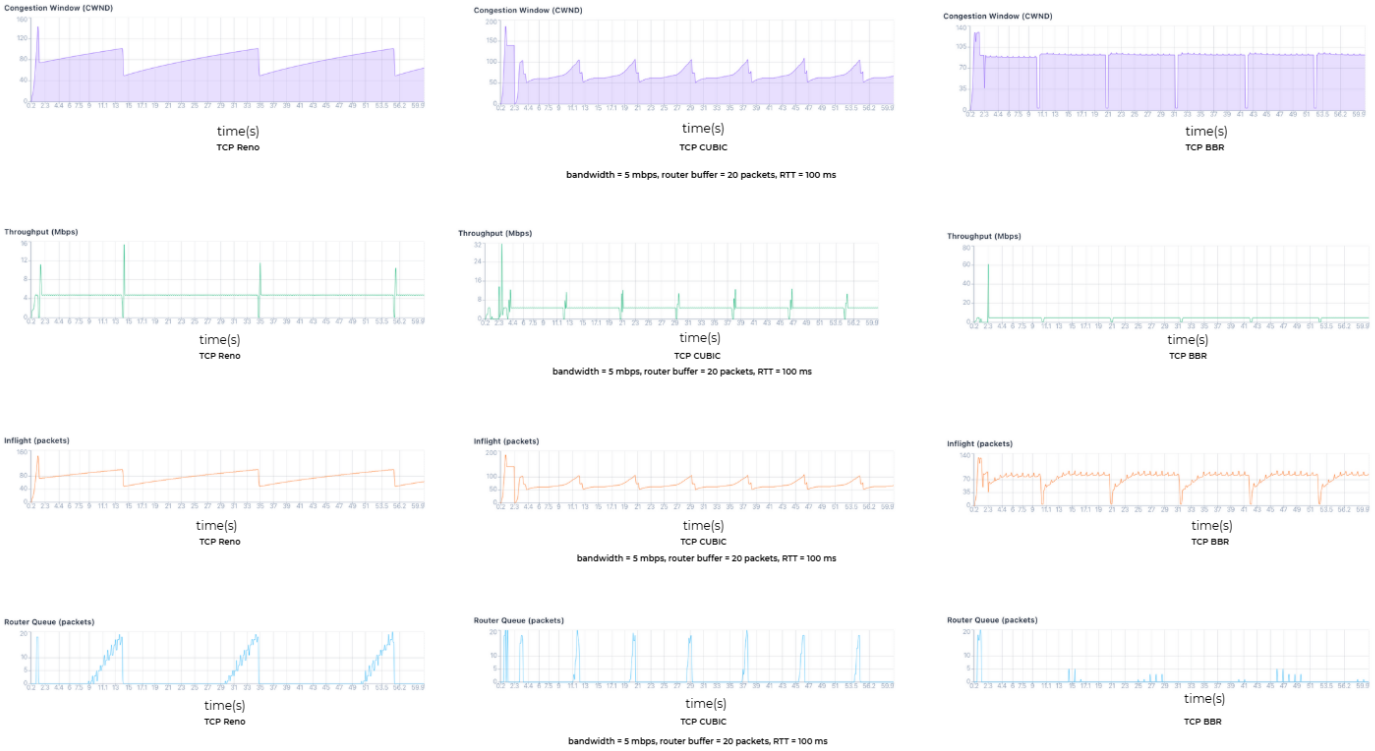


Figure 5: TCP (ns3) results

These graphs show the reasons why we had to turn to a Python mathematical modeling to simulate the TCP congestion control algorithms rather than use ns3's support. For example, in the throughput vs time graph for TCP Reno, the throughput shoots beyond the set bandwidth of 5 Mbps even though the congestion window just halved. In the throughput vs time graph for TCP BBR, the throughput is abysmally low. In these graphs, `ssthres` is internally set by ns3 to an arbitrarily high value initially and this seems to give correct graphs for cwnd vs time for all Reno, CUBIC and BBR. However, when we change this value to a lower one, these graphs break. This is further elaborated in the "Challenges" section below.

6.2 Multi Flow Topologies

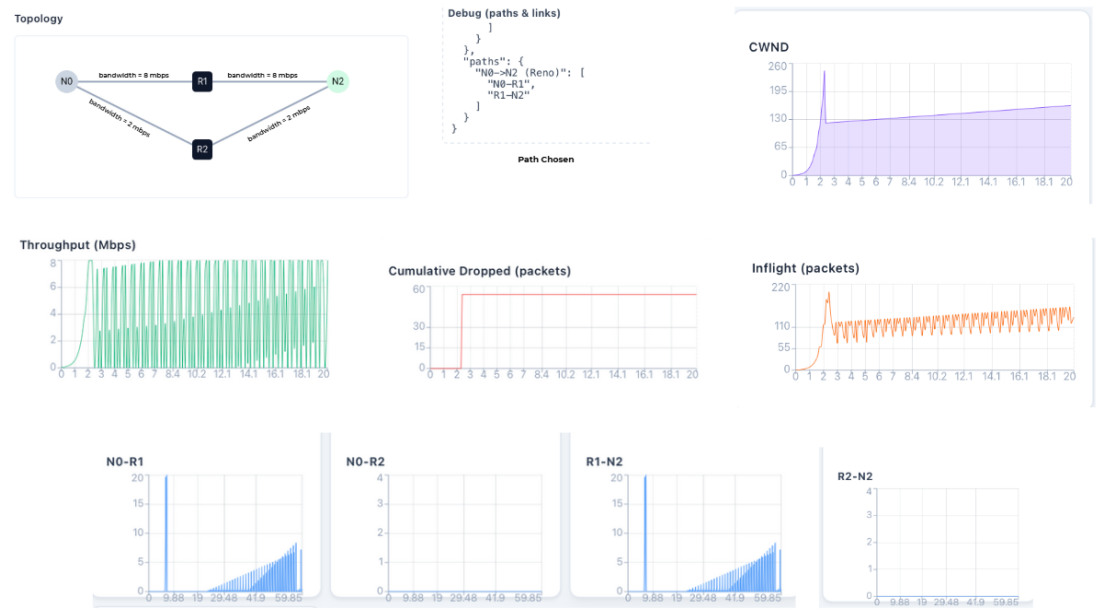


Figure 6: Multiflow (parallel) results for Reno

The given topology is a pair of routers connected in parallel with links from N0 → R1 → N2 which has a bandwidth of 8 Mbps and N0 → R2 → N2 which has a bandwidth of 8 Mbps. The path chosen by the packets for traversal from N0→N1 is N0 → R1 → N2.

- The throughput graph is fluctuating wildly. This is likely because the router buffer size is much smaller than the bandwidth-delay product. $BDP = Bandwidth(8Mbps) * RTT(100ms) = 0.8Mb = 100KB = 67packets$ (at 1500 bytes/packet). However, the buffer size is set to 20 packets, which is clearly very small. Therefore, the buffer fills up immediately, there is massive packet loss and throughput drops to zero repeatedly. This demonstrates how setting wrong parameters would affect the output of the network. It is possible that bufferbloat may be observed in the case of Reno and CUBIC if buffer size is increased massively.
- Due to the higher bandwidth of the upper link, packets prefer the upper route. This makes the link queue of R2 empty for all time, as can be seen in the graphs in the bottom row.

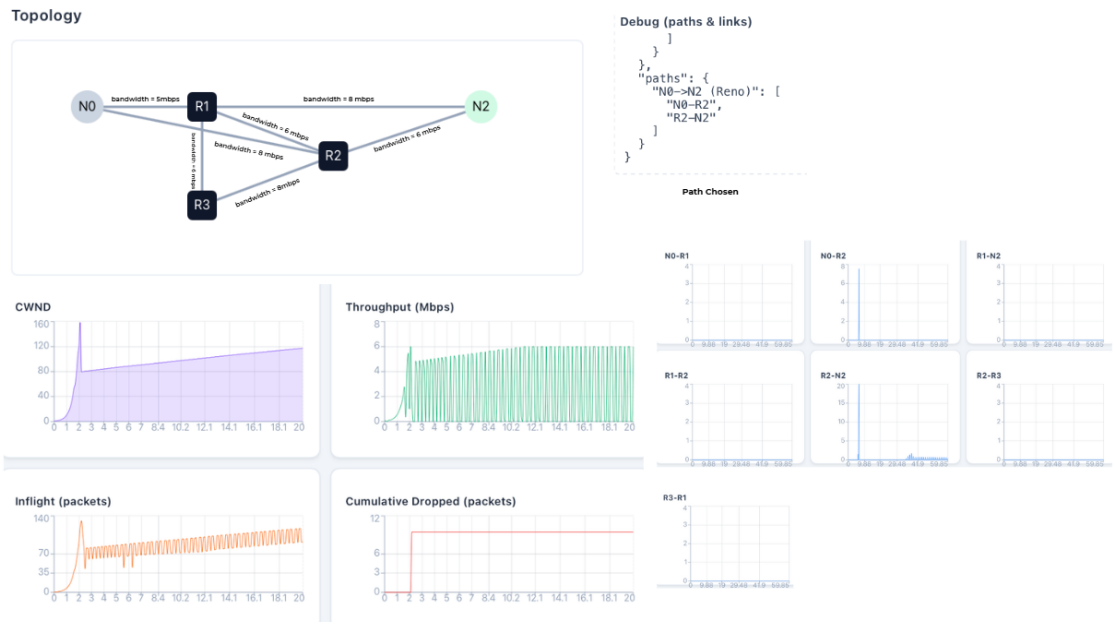


Figure 7: Multiflow (triangle) results for Reno

The given topology consists of 3 routers connected in a triangle with links from N0 → R1 (bandwidth = 5 Mbps), N0 → R2 (bandwidth = 8 Mbps), N2 → R1 (bandwidth = 8 Mbps), N2 → R2 (bandwidth = 6 Mbps), R1 → R2 (bandwidth = 6 Mbps), R1 → R3 (bandwidth = 6 Mbps) and R2 → R3 (bandwidth = 8 Mbps). The path chosen by the packets for traversal from N0 → N2 is N0 → R2 → N2.

- Activity in the link queues of the routers can only be seen for N0-R2 and R2-N2.
- Again, the same problem of buffer size being too small plagues the throughput vs time graph in this topology.

6.3 Challenges Faced and Future Work

This project was not without several technical challenges.

1. Firstly, while setting up the NS-3 simulation environment, several challenges were encountered that affected the accuracy and consistency of the results. Due to these limitations and the complexity involved in debugging the simulation environment, we focused more on validating our results through the Python-based simulation framework, which provided faster iteration and clearer, reproducible outcomes. Nonetheless, resolving the NS-3 configuration inconsistencies remains part of future work to ensure complete alignment between both simulation approaches.
2. Another tool we considered was using NAM (Network Animator), a visualization tool for NS-2 and NS-3, to graphically represent the topology of nodes, routers, and data flow during the simulation. However we had already gone forward with a React based graph for single flow and therefore decided to continue with it for multi-flow as well.
3. Clearly the simulation of multi-flow network was not as desired. We do wish to look into it in the future and fix the simulations if time permits.

7 Conclusion

An interactive TCP congestion control visualizer was developed to demonstrate the behavior of Reno, CUBIC, and BBR algorithms under various network conditions. Through both Python-based mathematical modeling and ns-3 packet-level simulation, we created an educational tool that provides real-time visualization of key metrics including congestion window evolution, throughput, inflight packets, and packet loss.

The implementation revealed important insights into TCP dynamics: Reno's characteristic sawtooth pattern and aggressive response to loss, CUBIC's time-based cubic growth function optimized for high-bandwidth networks, and BBR's model-based approach that maintains stable throughput without relying on packet loss signals. Multi-flow experiments highlighted the critical importance of proper parameter configuration, particularly the relationship between buffer size and bandwidth-delay product, demonstrating how undersized buffers can cause severe throughput oscillations.

While challenges with ns-3 configuration consistency led us to rely primarily on the Python simulation framework, the project achieved its core objective of creating an intuitive, web-based platform for understanding congestion control. The React-based frontend successfully communicates with the Flask backend to provide smooth, real-time graph animations that make complex networking concepts accessible to students and instructors.

Future work includes resolving ns-3 simulation inconsistencies, implementing more sophisticated routing algorithms for multi-flow scenarios, and perhaps integrating additional TCP variants such as Vegas or Illinois. Overall, this visualizer serves as a valuable educational resource for students like us that bridges the gap between theoretical TCP concepts and their practical behavior in simulated network environments.

References

[1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Pearson, 2021. ISBN: 978-0135928615.

[2] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 6th ed. Pearson, 2011. ISBN: 978-0132126953.

[3] nsnam.org, "16.5. TCP Models in ns-3," Model Library, Available: `https://www.nsnam.org/docs/models/html/tcp.html#congestion-control-algorithms`.

[4] L. Peterson and B. Davie, *"TCP Congestion Control"*, in *Computer Networks: A Systems Approach*, [Online]. Available: `https://book.systemsapproach.org/congestion/tcpcc.html`.

[5] ns-3 Project, "ns-3 Tutorial," [Online]. Available: `https://www.nsnam.org/docs/tutorial/html/`.

[6] D. Singh, "Experimenting with TCP Congestion Control," [Online]. Available: `https://dipsingh.github.io/TCP-Congestion-Experiment/`. Accessed: Nov. 12, 2025.

[7] "ns3 installation on Windows 11 — WSL," YouTube, Nov. 29, 2024. [Online]. Available: `https://www.youtube.com/watch?v=3U_sKkbSMjQ`. Accessed: Nov. 12, 2025.

[8] R. Kandoi, "TCP-Congestion-Control-in-NS2" GitHub repository, 2025. [Online]. Available: `https://github.com/Rishabhkandoi/TCP-Congestion-Control-in-NS2`.

[9] Facebook Open Source, "React – A JavaScript library for building user interfaces (Version 18.x)" Meta Platforms, Inc. [Online]. Available: `https://react.dev/learn`.

[10] Recharts Contributors, "Recharts – A composable charting library built on React components". [Online]. Available: `https://recharts.org/`.

[11] Pallets Projects, "Flask – Web development, one drop at a time (Version 3.x)" Pallets. [Online]. Available: `https://flask.palletsprojects.com/`

[12] Flask-CORS Contributors, "Flask-CORS – A Flask extension for handling Cross Origin Resource Sharing (CORS)". [Online]. Available: `https://flask-cors.readthedocs.io/en/latest/api.html`.