Network Performance Using Different TCP Congestion Control Algorithms

CS 244: Computer Networks

Lamar AlSubhi

Methodology

To evaluate the behavior of different TCP congestion control algorithms, I designed and executed a series of controlled experiments. The goal was to measure how each TCP flavor behaves in response to different link types and traffic conditions in terms of throughput, latency, packet loss, and congestion window size. All experiments were scripted, logged for consistency, and focused on four TCP flavors: BBR, CUBIC, Reno, and Vegas. Traffic was generated with iperf3 from a client to a server, with supporting measurements collected using ping to calculate RTT and ICMP loss and ss -ti to get a snapshot of the congestion window state.

Setup

The setup consisted of two devices: an M4 MacBook Pro acting as the client and an HP OMEN laptop running Linux acting as the server. I maintained a runs.csv file to organize every variation of experiment, with columns:

- run_id – numeric identifier for the run
- scenario – baseline, light traffic, or heavy traffic
- link_setup – WiFi–WiFi or ETH–WiFi
- tcp_flavor – BBR, CUBIC, Reno, or Vegas
- background – none, light, or heavy background load (same as traffic)
- bidir – indicates whether traffic was unidirectional or bidirectional (if any)
- trial – label for trial A or B

I designed this structure to ensure that every combination of conditions was covered systematically and repeated at least twice. The full set of runs is summarized below:

Scenario	Link setups	TCP flavors	Trials	Runs
Baseline	WiFi–WiFi, ETH–WiFi	BBR, CUBIC, Reno, Vegas	A, B	16
Light Background	WiFi–WiFi, ETH–WiFi	BBR, CUBIC, Reno, Vegas	A, B	16
Heavy Background	WiFi–WiFi, ETH–WiFi	BBR, CUBIC, Reno, Vegas	A, B	16
Bonus Bidirectional	WiFi–WiFi, ETH–WiFi	BBR, CUBIC	A, B	8
Total	—	—	—	**56**

Procedure

I automated the experiments with a sequence of Python scripts:

1. run_test.py

    - Executes a single run from runs.csv with a given run_id
    - Produces raw logs:
        - {run_id}_iperf.json - iperf3 throughput and retransmission statistics
        - {run_id}_rtt.txt - ping RTT samples and ICMP loss statistics
        - {run_id}_cwnd.txt - per-second socket snapshots showing CWND
          {run_id}_meta.txt - environment details to confirm active TCP flavor

2. analysis.py

    - Parses raw logs into time-series CSVs:
        - {run_id}_throughput.csv
        - {run_id}_rtt.csv
        - {run_id}_cwnd.csv
    - Extracts summary statistics
        - mean, p90, p95 for both throughput and RTT
        - ICMP loss percentage
        - median/p95 CWND
    - Appends one summary row to results.csv
    - Generates per-run plots for throughput, RTT, and CWND

3. results_agg.py

    - Reads results.csv and combines repeated trials (A and B).
    - Produces results_agg.csv

4. summary.py

    - Generates comparison plots across algorithms and conditions (throughput, RTT, loss percent, CWND, and throughput vs RTT scatter)

This workflow provides a complete pipeline: from raw measurements, to per-run summaries, to aggregated results, and finally to comparative figures.

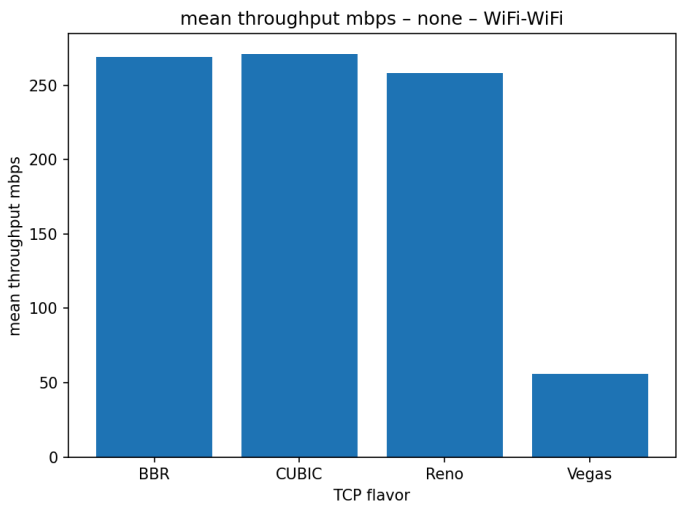
Topology

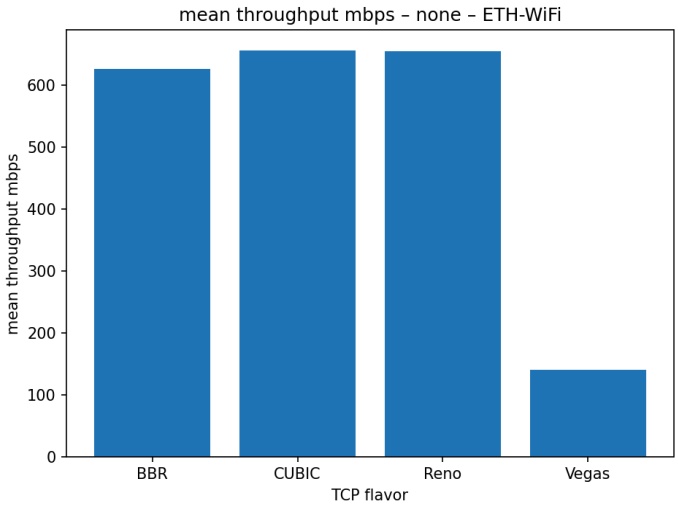
I used a simple client-server arrangement:

- Receiver (server): MacBook Pro (Wi-Fi)
- Sender (client): HP Omen laptop running Linux (connected alternately via Wi-Fi or Ethernet)
- Access point: both devices connected to the same consumer Wi-Fi router.

Results & Analysis

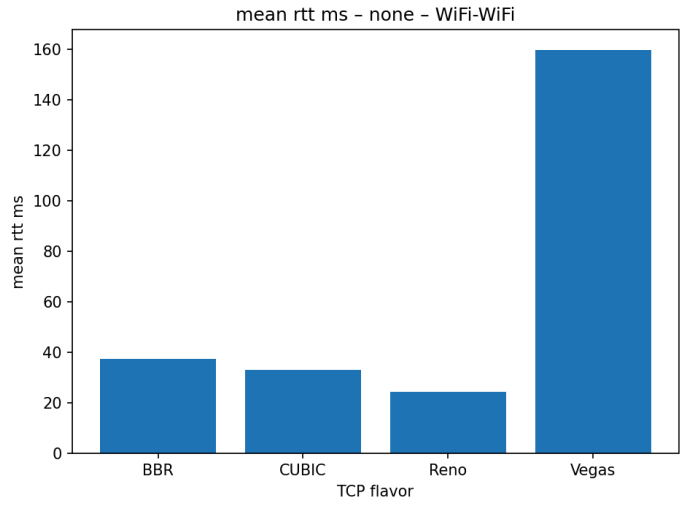
Throughput

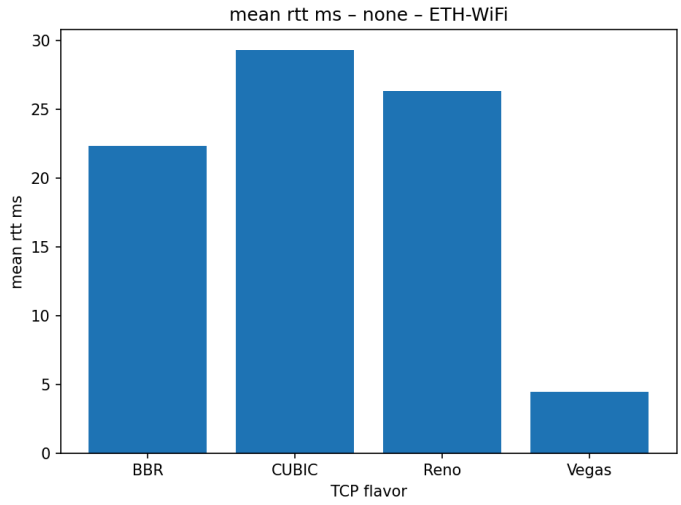




The ETH–WiFi setup consistently showed higher throughput than WiFi–WiFi across all four algorithms, which reflects the more stable and less congested wired path. Within both connection types, BBR, CUBIC, and Reno delivered similar high throughput, while Vegas consistently underperformed by a large and clear margin. This shows how the delay-based control in Vegas limits its ability to fully utilize available bandwidth, especially in comparison to the loss-based and pacing-based algorithms.

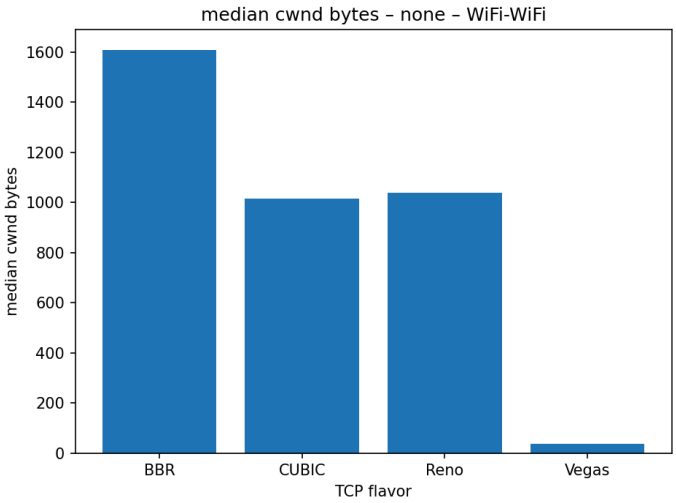
RTT

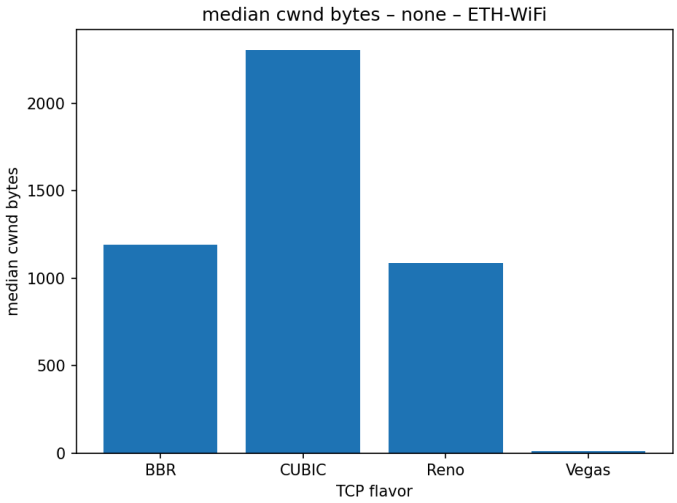




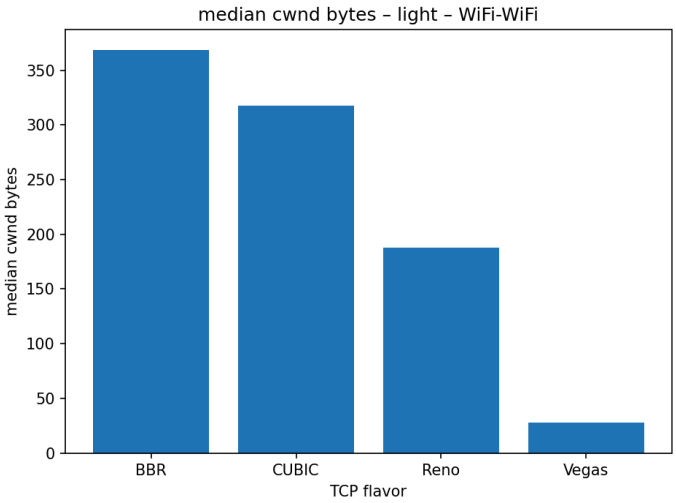
On the WiFi–WiFi setup, while BBR, CUBIC, and Reno maintained average RTTs between 25–40 ms, Vegas performed poorly. This is explained by how the delay-based control loop can react badly to wireless variability, causing the average RTT to inflate to 160 ms. On the Ethernet–WiFi setup, all three loss-based algorithms (BBR, CUBIC, Reno) showed RTTs in the 20–30 ms range while Vegas achieved the lowest delay overall (4 ms). This shows how delay-based approaches thrive in stable wired environments where queueing delay is the dominant signal. So while Vegas is capable of keeping delay low on clean links, its sensitivity to noise and variability on Wi-Fi makes it less than ideal for wireless conditions.

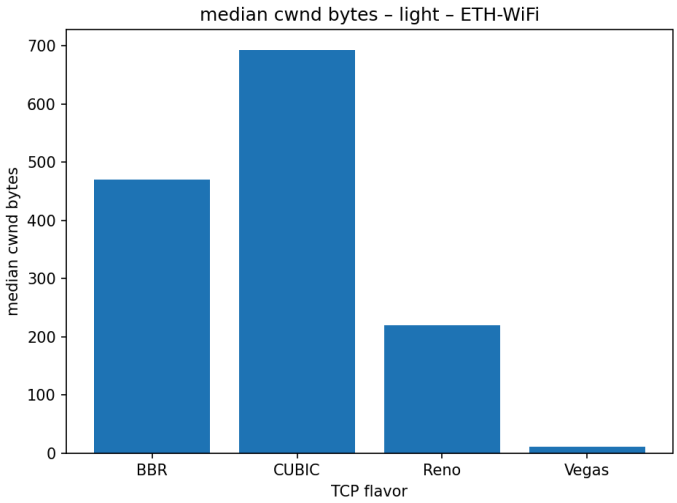
CWND



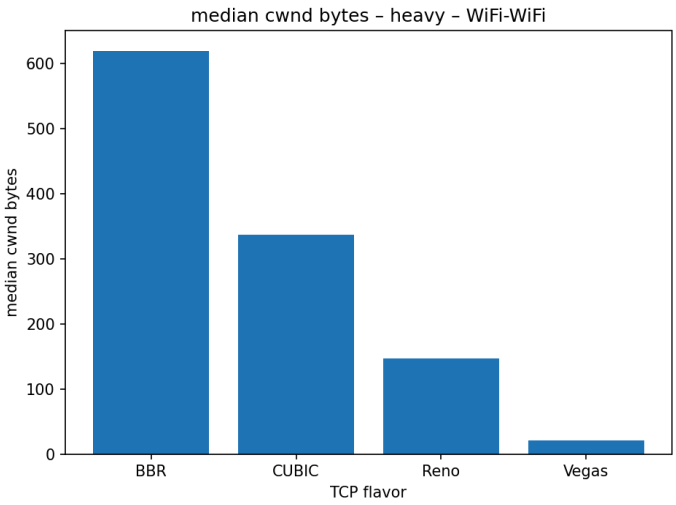


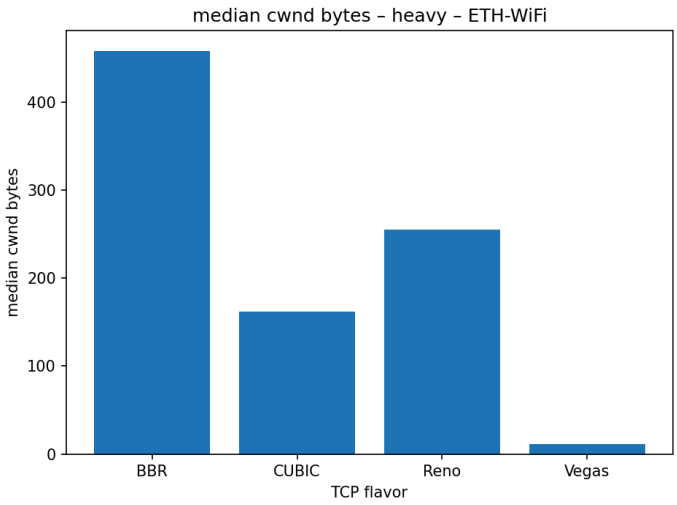
Baseline: In WiFi–WiFi runs, BBR sustained the largest cwnd, while in ETH–WiFi runs CUBIC grew the largest. Reno maintained smaller windows in both cases, and Vegas consistently underperformed with nearly negligible cwnd values.





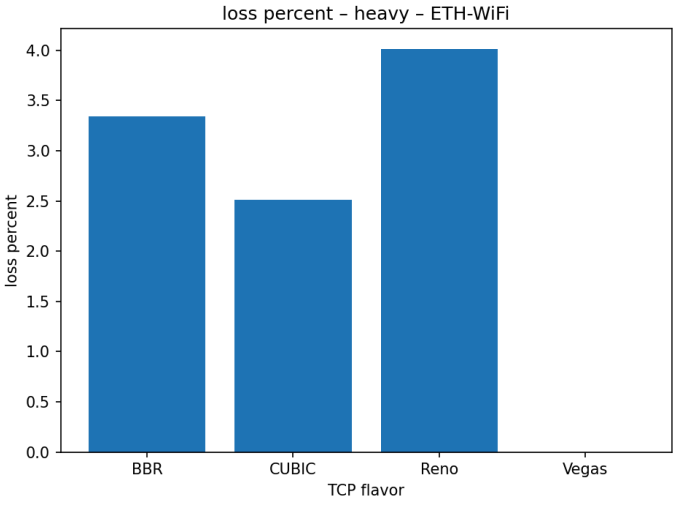
Light background: Introducing light contention reduced cwnd slightly, but the relative ranking remained the same: BBR and CUBIC at the top, Reno in the middle, and Vegas at the bottom. This suggests that light traffic does not fundamentally disrupt the algorithms.

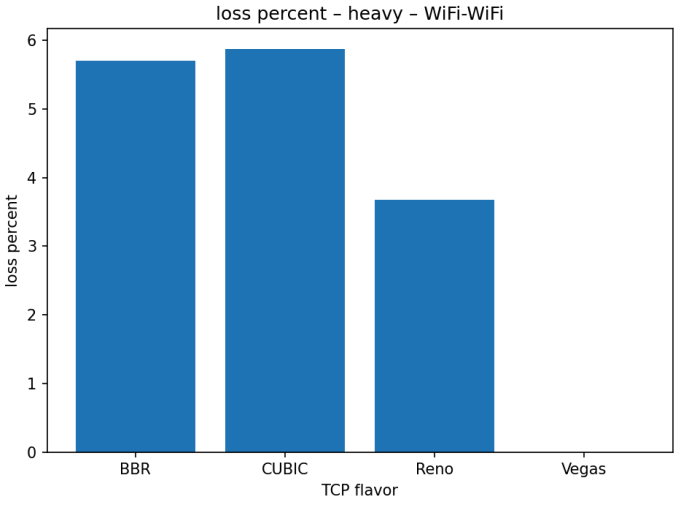




Heavy background: With heavy traffic, cwnd values dropped sharply across all flavors. BBR remained the most resilient, keeping the largest window in both link setups. CUBIC's cwnd shrank considerably, Reno even more, and Vegas stayed minimal. Heavy traffic certainly had more of an effect as it caused CUBIC to lose its ETH-WiFi lead.

Loss Performance





This is where Vegas's safe and slow approach shines. On WiFi–WiFi, BBR and CUBIC experienced the highest loss (~5–6%), reflecting their aggressive attempts to keep the link fully utilized. Reno showed moderate loss (~3–4%), consistent with its more conservative sawtooth behavior. Vegas reported virtually no loss, but only because it had already reduced its sending rate drastically, which explains its very low throughput in earlier figures. On ETH–WiFi, the overall loss percentages were lower but the same pattern held: Reno and CUBIC balanced throughput and loss, BBR tolerated higher loss in order to maintain throughput, and Vegas avoided loss but at the cost of utilization.

Conclusion & Recommendations

From the experiments, CUBIC emerges as the best all-around TCP congestion control algorithm for my testbed. It consistently achieved the highest throughput, especially under Ethernet-to-Wi-Fi links, while maintaining acceptable RTT and packet loss. Compared to BBR, it was less susceptible to large RTT increases under stress; compared to Reno, it better utilized available bandwidth; and compared to Vegas, it avoided the severe underutilization seen on wireless links.

While no single flavor is universally optimal, my results show that CUBIC strikes the best balance between throughput, delay, and robustness across diverse conditions, making it the most suitable general-purpose choice.

That said, certain scenarios may still favor alternatives:

- For high-throughput streaming over Wi-Fi, BBR is a winner since it pushes the link harder even at the cost of loss.
- For latency-sensitive gaming on Ethernet, Vegas offers the lowest delay, provided the link is stable.

Scripts & More

All scripts used in this project (run_test.py, analysis.py, results_agg.py, and summary.py) are hosted in my GitHub repository:

https://github.com/LamarAlSubhi/CS244/tree/main/as2

The repository includes raw log examples, CSV outputs, and other plots. The repository was kept private and made public only for submission to ensure no other students had access.