The Libyan Academy of Graduate Studies – Misrata Branch
Department of Computer Science

TCP Congestion Control Performance under Starlink-like Impairments
(Replication Study with Linux Namespaces)

Prepared by: Abobker Abdullah Alzwai

Supervised by Dr. Farij Ehtiba

Fall 2025

Abstract

Satellite Internet links such as Starlink exhibit characteristics that can confuse traditional loss-based TCP congestion control, including non-congestion packet loss and rapidly changing round-trip time (RTT). This report replicates a Starlink-inspired evaluation by emulating a client–server path in Ubuntu 22.04 using Linux network namespaces, iperf3 traffic generation[1], and Linux traffic control (tc) with TBF rate limiting and netem delay/loss[2]. Four experiment sets were conducted: (i) baseline (unconstrained) throughput for 14 TCP congestion control algorithms (CCAs), (ii) Starlink-like impairment (10 Mbit/s, 50 ms delay, 2% loss) with single-flow and 10-flow tests, and (iii) a two-flow fairness test where BBR and CUBIC compete simultaneously. Results show that in the impaired single-flow scenario, BBR achieves substantially higher throughput than CUBIC (≈5.96 vs. 1.97 Mbit/s), indicating improved robustness to random loss. With 10 parallel flows, BBR remains competitive (≈7.64 Mbit/s) while CUBIC approaches similar utilization (≈6.00 Mbit/s). Fairness measurements suggest BBR does not completely starve CUBIC in this setup, but it can take a modestly larger share when it starts earlier. The findings support using model-based congestion control as a practical choice for Starlink-like conditions, while highlighting limitations of loss-driven algorithms on lossy paths[5].

Keywords: TCP, congestion control, BBR, CUBIC, Starlink, satellite networks, traffic control, netem, iperf3.

1. Introduction

Transmission Control Protocol (TCP) is responsible for the majority of Internet data transfer, and its congestion control algorithm (CCA) determines how quickly a sender injects packets into the network. Most classic CCAs interpret packet loss as a primary signal of congestion; when loss occurs, they reduce the sending rate to alleviate queue overflow [3]. This assumption works well on many terrestrial wired networks, but it becomes problematic for satellite Internet systems where packet loss can be caused by channel variations, link-layer effects, or handoffs rather than router queue congestion.

Starlink is a low Earth orbit (LEO) satellite network designed to provide broadband connectivity with lower latency than traditional geostationary satellite systems. However, LEO links can still exhibit highly variable RTT, transient bottlenecks, and loss events not tied to congestion. In such conditions, a loss-based CCA may repeatedly back off even when the path is not congested, underutilizing available capacity.

This report focuses on comparing two widely discussed CCAs—CUBIC and BBR—under Starlink-like impairments. CUBIC is the default CCA on many Linux systems and is standardized in RFC 8312 [4].BBR (Bottleneck Bandwidth and RTT) is a model-based algorithm that estimates the bottleneck bandwidth and minimum RTT and paces packets accordingly [5].

The study is implemented using a controlled emulation environment and produces quantitative throughput and fairness results.

2. Problem Statement

The core problem addressed is that loss-based TCP congestion control may misinterpret non-congestion loss on satellite-like links as congestion, causing unnecessary rate reductions and poor throughput. The objective is to evaluate whether a model-based algorithm (BBR) maintains higher throughput than CUBIC under Starlink-like impairments, and to examine how fairly BBR shares bandwidth when competing with CUBIC on the same bottleneck.

3. Objectives

1) Build a reproducible client–server testbed using Linux network namespaces and a veth pair.

2) Emulate Starlink-like path impairments (bandwidth limit, delay, and packet loss) using tc/TBF and netem.

3) Measure end-to-end TCP throughput for multiple congestion control algorithms using iperf3.

4) Compare single-flow and multi-flow behavior (P=1 vs. P=10) to assess link utilization.

5) Evaluate pairwise fairness between BBR and CUBIC under the same bottleneck using overlapping iperf3 sessions.

6) Generate visual summaries (bar charts) and interpret the outcomes with networking principles.

4. Methodology

This section describes the experimental environment, the network emulation model, and the measurement procedure. The full command scripts are provided in the Appendices.

4.1 Testbed and Tools

All experiments were conducted on Ubuntu 22.04 (Linux kernel 6.8.x) inside a VirtualBox virtual machine. The testbed uses two Linux network namespaces: ns_client and ns_server. A virtual Ethernet (veth) pair connects the namespaces, giving an isolated point-to-point topology that behaves like a simple client–server link. The ip tool (iproute2) is used to create and manage namespaces, while iperf3 generates TCP traffic and reports receiver throughput. Linux traffic control (tc) is used to apply queuing disciplines (qdiscs) to the client-side interface [1] [2].

Figure 1 illustrates the emulated topology used in this work.

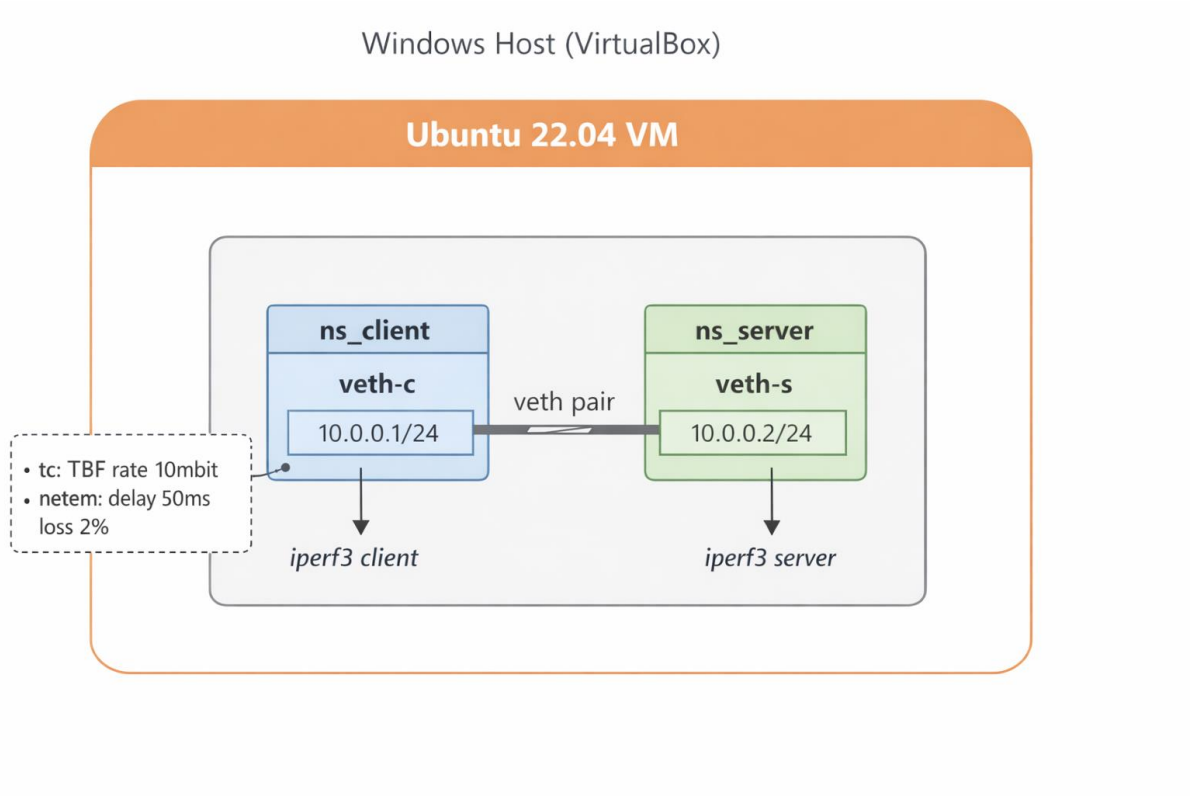

Figure 1. Emulated client–server topology using Linux namespaces and veth. Impairments are applied on the client egress (veth-c).

4.2 Emulation of Starlink-like Impairments

To approximate satellite conditions, the client egress interface (veth-c) is configured with a two-stage qdisc:

• Token Bucket Filter (TBF) to enforce a bottleneck rate of 10 Mbit/s.

• netem to add 50 ms one-way delay (effectively increasing RTT) and 2% random packet loss[2].

This impairment model intentionally introduces loss that is not tied to queue overflow, mimicking the key challenge for loss-based CCAs: distinguishing random loss from congestion. In this implementation, delay is fixed and does not include jitter; the impact of jitter is not evaluated and is discussed as a limitation.

4.3 Congestion Control Algorithms Tested

Fourteen CCAs available in the Linux kernel were enabled via kernel modules and tested: bbr, cubic, bic, cdg, dctcp, highspeed, htcp, hybla, illinois, lp, nv, scalable, veno, and westwood. Each experiment explicitly selects the algorithm using iperf3's -C option to avoid global side effects[1].

4.4 Measurement Procedure

For each CCA, throughput was measured as the iperf3 'SUM receiver' bitrate[1]. Two traffic patterns were used:

• Single flow (P=1): captures how a single TCP connection behaves.
• Ten parallel flows (P=10): estimates how well the algorithm can fill the bottleneck in aggregate.

For the impairment model, the test duration was long enough (t=180 s in the three-CCA baseline and t=10 s for the 14-CCA sweep used to produce the provided charts) to reach steady behavior. A fairness experiment was also performed by starting one algorithm as a 'primary' 60-second transfer and then starting a 'secondary' 30-second transfer of the competing algorithm halfway through, on separate iperf3 server ports. Bandwidth share was computed from the final receiver rates of the overlapping transfers.

5. Results and Analysis

This section summarizes measured throughput under (i) normal conditions (no imposed bottleneck) and (ii) impaired conditions (10 Mbit/s, 50 ms delay, 2% loss). The complete dataset is stored in the file results_14cc_fixed2.csv and is summarized using figures and tables below[7].

5.1 Key Comparison: BBR vs. CUBIC

Table 1 compares BBR and CUBIC across the two conditions and two flow counts. Under the impaired condition, BBR maintains significantly higher throughput for a single flow, while the gap narrows with 10 parallel flows.

Condition	Flows (P)	BBR Throughput	CUBIC Throughput	BBR Gain vs. CUBIC
Normal (no shaping)	1	7.39 Gbit/s	35.06 Gbit/s	0.21×
Normal (no shaping)	10	13.91 Gbit/s	27.86 Gbit/s	0.50×
Impaired (10M,50ms,2% loss)	1	5.96 Mbit/s	1.97 Mbit/s	3.03×
Impaired (10M,50ms,2% loss)	10	7.64 Mbit/s	5.96 Mbit/s	1.28×

Table 1. Throughput comparison between BBR and CUBIC (values derived from results_14cc_fixed2.csv).

5.2 Throughput of 14 CCAs (Bar Charts)

Figures 2–5 visualize throughput across the 14 tested CCAs for normal and impaired conditions.

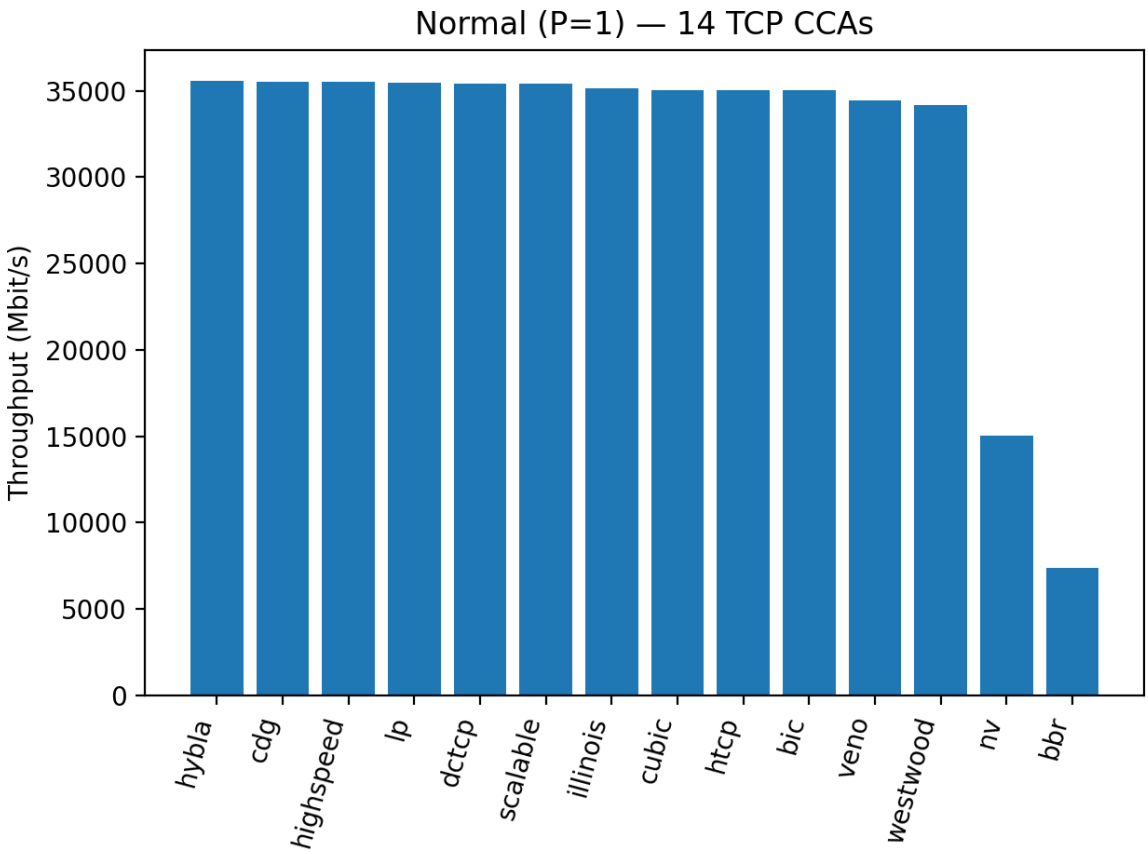

Figure 2. Normal condition (P=1): throughput for 14 TCP CCAs.

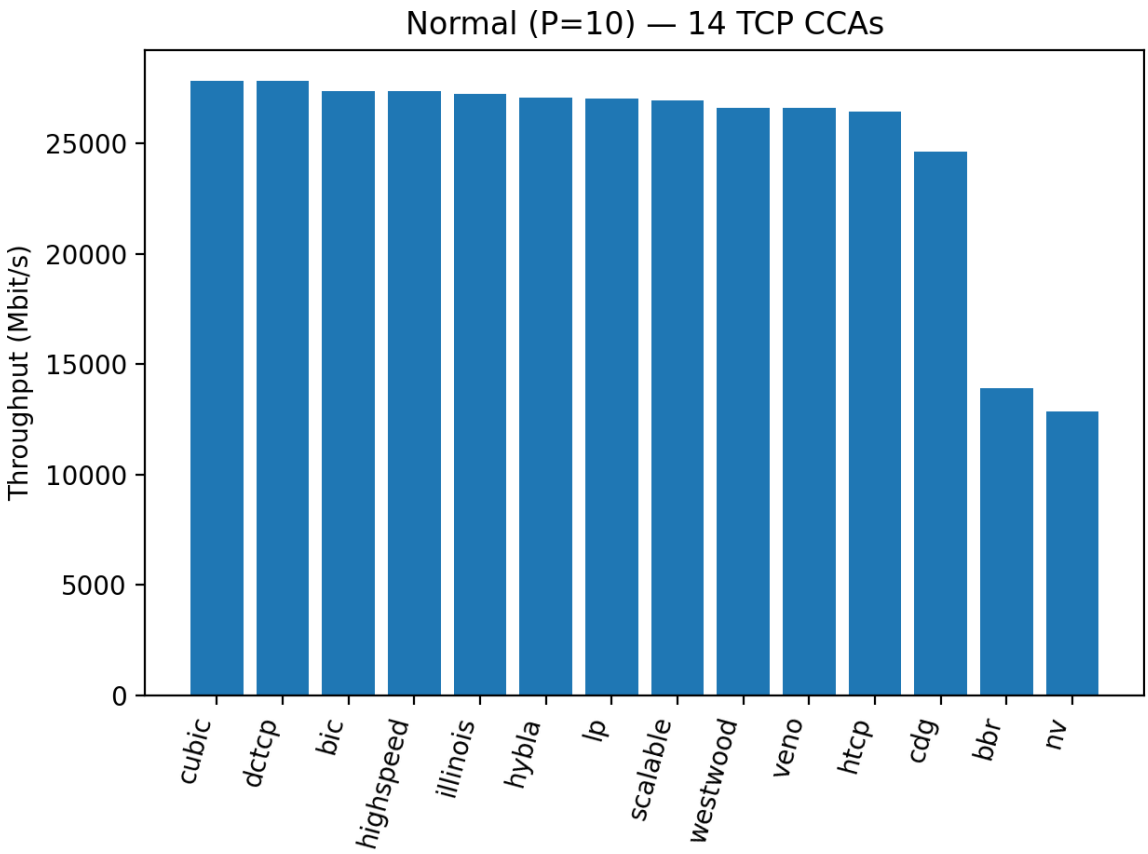

Figure 3. Normal condition (P=10): throughput for 14 TCP CCAs.

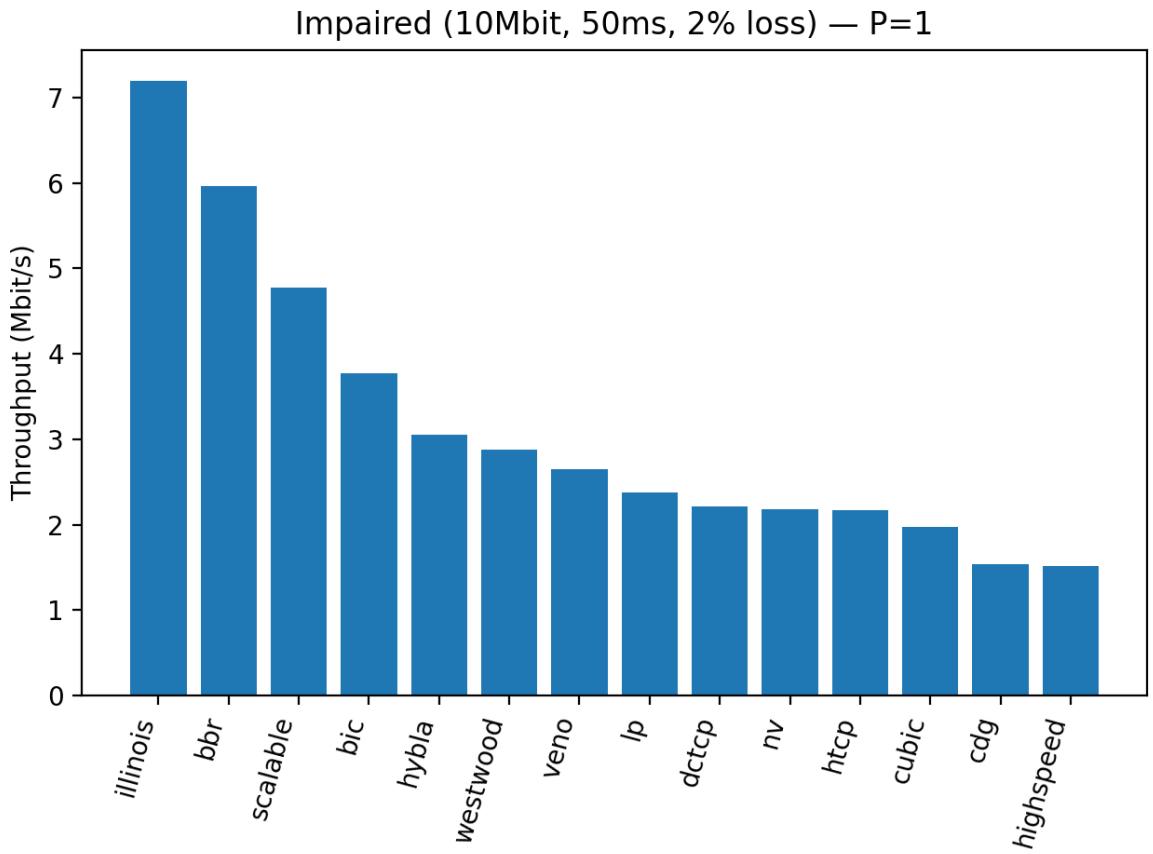

Figure 4. Impaired condition (P=1): throughput for 14 TCP CCAs (10 Mbit/s, 50 ms, 2% loss).

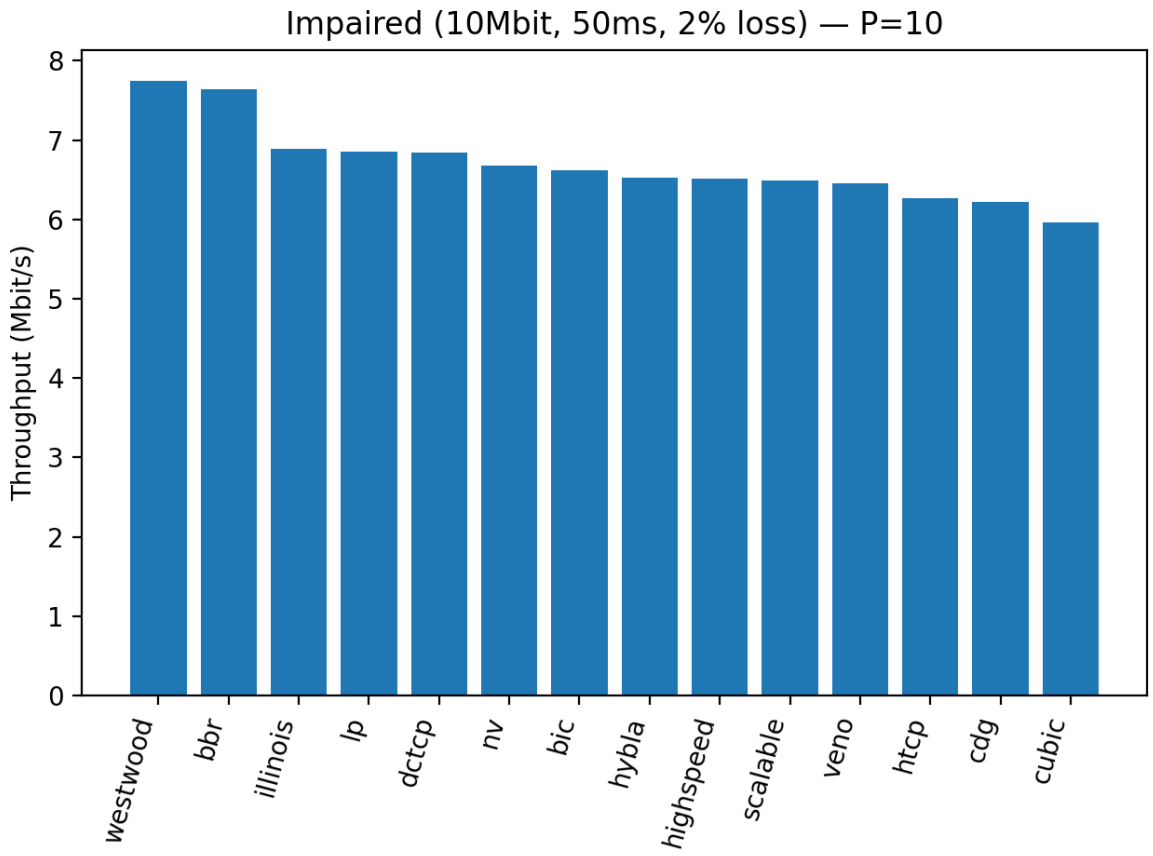

Figure 5. Impaired condition (P=10): throughput for 14 TCP CCAs (10 Mbit/s, 50 ms, 2% loss).

5.3 Top Performers

Table 2 lists the top five CCAs by receiver throughput for each scenario. In the impaired single-flow case, Illinois and BBR lead; in the impaired multi-flow case, Westwood slightly leads BBR.

Scenario	1st	2nd	3rd–5th (descending)
Normal, P=1	hybla (35.59 Gbit/s)	cdg (35.52 Gbit/s)	highspeed (35.50 Gbit/s); lp (35.48 Gbit/s); dctcp (35.42 Gbit/s)
Normal, P=10	cubic (27.86 Gbit/s)	dctcp (27.84 Gbit/s)	bic (27.38 Gbit/s); highspeed (27.38 Gbit/s); illinois (27.26 Gbit/s)
Impaired, P=1	illinois (7.20 Mbit/s)	bbr (5.96 Mbit/s)	scalable (4.78 Mbit/s); bic (3.77 Mbit/s); hybla (3.05 Mbit/s)
Impaired, P=10	westwood (7.75	bbr (7.64 Mbit/s)	illinois (6.89 Mbit/s);

	Mbit/s)	lp (6.86 Mbit/s); dctcp (6.84 Mbit/s)

Table 2. Top five CCAs by throughput in each scenario.

5.4 Fairness Experiment: BBR vs. CUBIC Competition

To test bandwidth sharing, two overlapping transfers were executed. In Case A, BBR ran as the primary 60 s flow while CUBIC started after 30 s for 30 s. In Case B, the order was reversed. Table 3 summarizes the receiver rates and computed shares.

Case	Primary	Secondary	Primary rate (Mbit/s)	Secondary rate (Mbit/s)	BBR share (%)
A	BBR (60 s)	CUBIC (30 s)	6.07	4.32	58.4
B	CUBIC (60 s)	BBR (30 s)	4.96	5.15	50.9

Table 3. Fairness results from overlapping BBR and CUBIC transfers (receiver rates from iperf3 logs).

Observation: In this environment, BBR tends to obtain a slightly larger share when it starts earlier (≈58% vs. 42%). When CUBIC starts earlier, the share becomes close to even (≈51% vs. 49%).

6. Discussion

The results align with a well-known conceptual difference between loss-based and model-based congestion control.

1) Loss sensitivity: CUBIC increases its congestion window until loss indicates congestion, then reduces the window and slowly increases again. On a path where 2% loss is random (not congestion), CUBIC repeatedly triggers backoff, so its average sending rate drops below the true bottleneck capacity. This behavior is consistent with the TCP congestion control principles described in RFC 5681 and the CUBIC specification (RFC 8312) [2], [3].

BBR, in contrast, estimates bottleneck bandwidth and minimum RTT and paces packets near the estimated bandwidth rather than relying on loss as the primary signal [4]. Therefore, when losses occur that do not reflect queue overflow, BBR is less likely to reduce its rate drastically, which explains its higher throughput under impairment (≈3× better than CUBIC for P=1).

2) Startup / HyStart effect: Modern Linux CUBIC includes improved slow-start behavior (HyStart/HyStart++ style heuristics) that try to detect incipient congestion earlier than classic slow-start to avoid large queueing delay. On satellite-like links with variable RTT, these heuristics can mis-detect congestion and exit slow-start prematurely, delaying ramp-up. HyStart++ is documented in RFC 9406 [6]. This can contribute to the lower initial aggressiveness of CUBIC compared to algorithms that maintain pacing models.

3) Single flow vs. multiple flows: With ten parallel flows, overall utilization improves for many

algorithms because independent flows can statistically overcome loss events and fill the pipe. This reduces the advantage of any single algorithm and explains why CUBIC approaches BBR in the impaired P=10 case.

4) Why some CCAs outperform BBR in the sweep: In the impaired scenarios, Illinois and Westwood sometimes exceed BBR. Both include mechanisms intended to improve performance over lossy or wireless links, which may fit the chosen impairment model. This does not contradict the focus of the assigned study (BBR vs. CUBIC) but suggests that 'best algorithm' can depend strongly on the exact loss/delay/bandwidth regime.

5) Fairness: The fairness experiment indicates that BBR can take a modestly larger share when it is the incumbent flow. This is a known concern for model-based CCAs: because they try to maintain pacing at estimated bandwidth, they may not respond to congestion signals in the same way as loss-based flows, potentially disadvantaging them. However, in this setup BBR did not completely starve CUBIC, and the shares were near 50/50 in Case B.

Limitations: (i) the impairment model used fixed delay without jitter; (ii) the 14-CCA sweep used short runs (t=10 s) to generate comparative charts, which may not fully capture long-term dynamics; (iii) the topology is a single bottleneck applied at the client egress rather than a full end-to-end satellite path with handoffs and queueing at multiple points. These constraints should be stated during the discussion.

7. Conclusion and Recommendations

This replication demonstrates that under Starlink-like random loss and added delay, BBR achieves higher single-flow throughput than CUBIC and remains competitive under multi-flow load. Therefore, for satellite-like environments where packet loss is not a reliable congestion signal, a model-based congestion controller is recommended.

Practical recommendation: If the goal is to maximize throughput and stability over a lossy, variable-RTT link, test BBR (or its newer variants) as the first candidate. If fairness with classic loss-based traffic is a critical requirement, additional evaluation should be performed, including longer runs, varying loss rates, and jitter models, and considering Active Queue Management (AQM) at the bottleneck.

Appendix A — Key Commands (Reproducibility)

A.1 Create namespaces, veth, and addressing (example):

```
sudo ip netns add ns_client
sudo ip netns add ns_server
```

```
sudo ip link add veth-c type veth peer name veth-s
sudo ip link set veth-c netns ns_client
sudo ip link set veth-s netns ns_server

sudo ip -n ns_client addr add 10.0.0.1/24 dev veth-c
sudo ip -n ns_server addr add 10.0.0.2/24 dev veth-s
sudo ip -n ns_client link set veth-c up
sudo ip -n ns_server link set veth-s up
sudo ip -n ns_client link set lo up
sudo ip -n ns_server link set lo up
```

A.2 Apply the Starlink-like impairment model (rate limit + delay + loss):

```
# Bottleneck at 10 Mbit/s using TBF
sudo ip netns exec ns_client tc qdisc add dev veth-c root handle 1: tbf rate
10mbit burst 32kbit latency 400ms

# Add delay and random loss using netem
sudo ip netns exec ns_client tc qdisc add dev veth-c parent 1:1 handle 10:
netem delay 50ms loss 2%
```

A.3 Enable and verify congestion control algorithms:

```
# Example: load modules (ignore errors if built-in)
sudo modprobe tcp_bbr tcp_cdg tcp_dctcp tcp_highspeed tcp_htcp tcp_hybla
tcp_illinois tcp_lp tcp_nv tcp_scalable tcp_veno tcp_westwood tcp_yeah

# List available CCAs (inside namespace or globally)
sudo ip netns exec ns_client sysctl net.ipv4.tcp_available_congestion_control
```

A.4 Run iperf3 tests:

```
# Start server (daemon) in ns_server
sudo ip netns exec ns_server iperf3 -s -D

# Single flow, choose CCA (example: bbr)
sudo ip netns exec ns_client iperf3 -c 10.0.0.2 -t 180 -P 1 -C bbr

# Ten parallel flows, choose CCA (example: cubic)
sudo ip netns exec ns_client iperf3 -c 10.0.0.2 -t 180 -P 10 -C cubic
```

A.5 Fairness test idea (two ports):

```
# Start two servers
sudo ip netns exec ns_server iperf3 -s -D -p 5201
sudo ip netns exec ns_server iperf3 -s -D -p 5202
```

```
# Primary flow (60 s)
sudo ip netns exec ns_client iperf3 -c 10.0.0.2 -p 5201 -t 60 -P 10 -C bbr >
primary_bbr.txt &

# After 30 s, start secondary (30 s)
sudo ip netns exec ns_client iperf3 -c 10.0.0.2 -p 5202 -t 30 -P 10 -C cubic >
secondary_cubic.txt &
```

Appendix B — Output Files

The following files were produced during the replication and used to generate results and figures:

• results_14cc_fixed2.csv

• chart_normal_p1.png

• chart_normal_p10.png

• chart_imp_p1.png

• chart_imp_p10.png

• primary_bbr.txt

• secondary_cubic.txt

• primary_cubic.txt

• secondary_bbr.txt

References

[1] Course-provided paper (PDF): “Performance of TCP Congestion Control over Starlink,” ANRW 2025 (provided by instructor).

[2] M. Allman, V. Paxson, and E. Blanton, “TCP Congestion Control,” RFC 5681, Sept. 2009.

[3] S. Ha, I. Rhee, and L. Xu, “CUBIC for Fast Long-Distance Networks,” RFC 8312, Feb. 2018.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” ACM Queue, vol. 14, no. 5, pp. 50–71, Oct. 2016.

[5] S. Islam, L. Welzl, and M. Welzl, “HyStart++: Modified Slow Start for TCP,” RFC 9406, July 2023.

[6] Linux Foundation, “tc-netem(8) — Network Emulator,” man7.org Linux man-pages, accessed Dec. 2025.

[7] Linux Foundation, “tc-tbf(8) — Token Bucket Filter,” man7.org Linux man-pages, accessed Dec. 2025.

[8] Linux Foundation, “namespaces(7) — Overview of Linux Namespaces,” man7.org Linux man-pages, accessed Dec. 2025.

[9] ESnet, “iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool,” iperf.fr/manpages, accessed Dec. 2025.