



SIMON FRASER UNIVERSITY

ENSC 833 NETWORK PROTOCOLS AND PERFORMANCE

Spring 2026

Dr. Ljiljana Trajkovic

FINAL PROJECT REPORT

QoS Enforcement for Bidirectional UDP-Based Control and Telemetry Traffic Using Packet Scheduling

Project web page	https://tonysingh7.github.io/ensc-833/
Name	Harsimranjeet Singh
SFU student ID	301289434
SFU e-mail	hsa97@sfu.ca
Team #	Team 6

Table of Contents

1. ABSTRACT.....	3
2. INTRODUCTION.....	3
3. SYSTEM DESIGN AND METHODOLOGY.....	4
3.1 Network topology and traffic model.....	4
3.2 Traffic classes and DSCP mapping.....	5
3.3 Queueing disciplines under test.....	5
3.4 Experimental scenarios.....	6
3.5 Metrics and data processing.....	6
4. RESULTS.....	7
4.1 Baseline overloaded scheduler comparison.....	7
4.2 Per-class behavior and trade-offs.....	10
4.3 Load sweep.....	12
4.4 Follower sweep.....	14
4.5 Fairness analysis.....	15
5. DISCUSSION AND CONCLUSIONS.....	18
6. REFERENCES.....	19
APPENDIX A. CODE LISTING.....	20

1. ABSTRACT

This project evaluates how packet scheduling can enforce Quality of Service (QoS) for bidirectional UDP traffic in a centralized control system. Using ns-3, a single orchestrator exchanges control traffic with multiple followers across a shared bottleneck while reverse-direction status, telemetry, bulk, and competing background traffic contend for the same constrained resource. Five queueing disciplines were compared: FIFO, PFIFO_FAST-style priority FIFO, strict-priority PRIO, FQ-PIE, and FQ-CoDel. The study further examined the effect of enabling DSCP marking and classifier-aware priority mapping. Under overload, FIFO produced severe control degradation, reaching 40.8 ms mean control delay and 61.4% control loss at the baseline comparison point, while FQ-PIE and FQ-CoDel held control delay to 8.9 ms with zero control loss. DSCP produced the largest benefit when used with priority schedulers: PFIFO_FAST-style and PRIO improved from 40.8 ms and 61.4% control loss to 8.9 ms and 0.0% loss, respectively. Load-sweep and follower-sweep experiments showed that fair-queue schedulers remained stable as offered background load and follower count increased, whereas FIFO and non-DSCP priority configurations degraded rapidly. The results show that queue selection is not a cosmetic implementation detail in UDP-based control systems; it directly determines whether latency-sensitive control traffic survives overload conditions.

2. INTRODUCTION

Many practical networked control systems favor UDP because it avoids connection setup, retransmission delay, and protocol-level congestion behavior that may interfere with tight update cycles. In a centralized orchestrator-and-device architecture, however, the same network path often carries multiple traffic classes with very different requirements. Small control packets require low delay and low loss, while reverse-direction telemetry and status updates are valuable but can tolerate somewhat lower priority. Bulk transfers and unrelated background traffic can safely absorb the remaining capacity.

This problem is especially relevant in embedded and industrial control settings, where a supervisory node may exchange frequent small control messages with multiple field devices while also collecting status and telemetry over the same shared path. In such systems, the concern is not simply raw throughput; it is whether the network can preserve bounded delay and acceptable loss for the classes that actually influence control behavior. That makes the queue at the bottleneck a first-order design concern rather than a background implementation detail [1], [2], [10].

This combination creates a scheduling problem rather than a simple transport-choice problem. UDP itself does not classify traffic, reserve bandwidth, or prevent queue buildup. If all packets are treated identically at a shared bottleneck, short control messages can become trapped behind larger or more numerous packets, which increases queueing delay and can eventually drive packet loss. The central question of this project is therefore whether packet scheduling can enforce practical QoS separation for bidirectional UDP control and telemetry traffic without changing the application transport.

The project title remains the same as in the proposal: QoS Enforcement for Bidirectional UDP-Based Control and Telemetry Traffic Using Packet Scheduling. The final implementation preserves the original idea of a simple, single-bottleneck ns-3 topology, but extends it into a more complete experimental study that includes scheduler comparison, offered-load sweeps, follower-count sweeps, DSCP-aware classification, and fairness analysis.

Classic work in transport, differentiated services, and queueing directly motivates this study. UDP provides minimal transport service and leaves reliability and congestion behavior to the application or network edge [2]. The Differentiated Services field provides a standardized way to carry class information in the IP header [3], while integrated-services work established the broader principle that differentiated service quality

requires explicit treatment in the network rather than a purely best-effort forwarding model [4]. Fair queueing, link-sharing, generalized processor sharing, and deficit-round-robin literature showed that the queue discipline itself can strongly shape delay and service distribution across competing flows [5]-[8]. More recent active queue management work emphasized queue-delay control [9], and recent industrial-IoT studies continue to show that scheduling policy remains central when real-time and best-effort traffic coexist [10]. The present study uses Jain fairness metrics to quantify service distribution [11] and implements the experiments in ns-3 [12].

3. SYSTEM DESIGN AND METHODOLOGY

3.1 Network topology and traffic model

The simulated system contains one orchestrator, two routers connected by a bidirectional bottleneck, a set of follower nodes, and two bidirectional background-traffic endpoints. The orchestrator connects to Router A through an access link, the followers connect to Router B through individual access links, and the queueing discipline under test is installed on both directions of the Router A–Router B bottleneck. Access links run at 100 Mbps with 1 ms propagation delay, while the bottleneck is configured at 10 Mbps with 5 ms propagation delay. In the baseline experiments, the queue size is fixed at 40 packets and the simulation time is 20 s. The implementation also exposes all major settings as command-line arguments so that the same program can be reused for matrix runs and parameter sweeps.

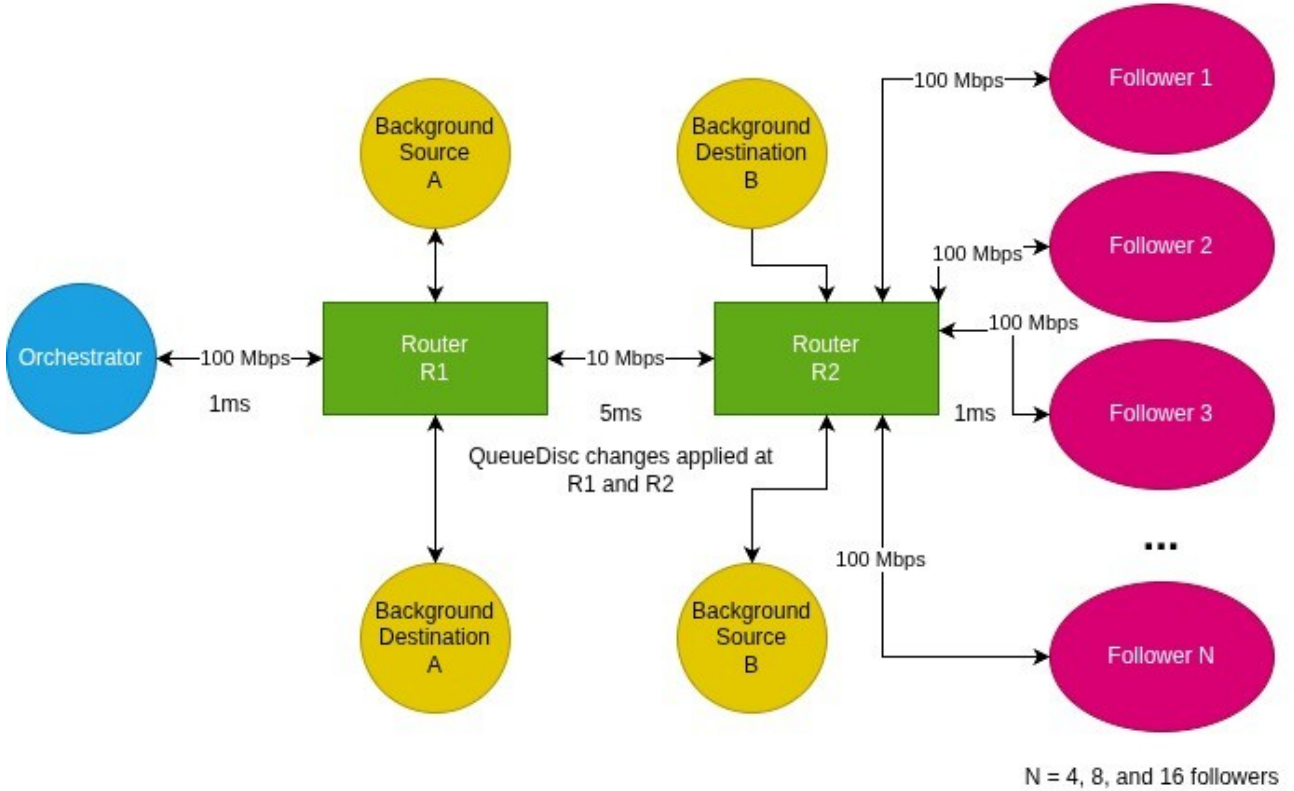


Figure 1. Simulation topology for bidirectional UDP control, telemetry, and background traffic.

Four application classes are generated within the control system itself. Control traffic is sent from the orchestrator to every follower. Status, telemetry, and bulk traffic are sent from each follower back to the orchestrator. Additional bidirectional background traffic is injected across the bottleneck in order to create overload conditions. This produces a bidirectional study rather than a purely downlink or uplink experiment, which is important because the forward and reverse directions carry different semantics and compete differently under shared queueing.

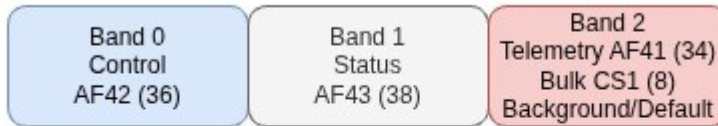
3.2 Traffic classes and DSCP mapping

Each application class is defined by a packet size, sending rate, port range, and DSCP value. Control traffic is marked AF42 when DSCP is enabled, status is marked AF43, telemetry is marked AF41, and bulk uses CS1. Background traffic is left at default treatment. These markings are important because the PFIFO_FAST-style and PRIO schedulers use explicit packet filters that map DSCP values into queue bands. AF42 is treated as the highest-priority control class; AF43 is placed below it; AF41 is placed lower still in the strict-priority configuration; and bulk or default traffic remains in the lowest-priority band.

At the IP layer, these values are carried in the Differentiated Services field defined for IPv4 and IPv6 headers in RFC 2474. In the present study, DSCP marking is used only to expose class identity to the packet filters used by the priority queueing cases; the marking does not itself reserve bandwidth, but it allows the queue discipline to separate control-relevant traffic from lower-priority traffic in a standards-consistent way [3].

Class	Direction	Packet size	Rate	DSCP	Priority intent
Control	Orchestrator -> followers	64 B	200 pps	AF42 (36)	Highest
Status	Followers -> orchestrator	96 B	100 pps	AF43 (38)	Medium
Telemetry	Followers -> orchestrator	512 B	50 pps	AF41 (34)	Lower
Bulk	Followers -> orchestrator	1400 B	20 pps	CS1 (8)	Lowest / filler
Background	Both directions	1400 B	Variable	Default	Competing load

PFIFO FAST-style 3-band mapping



PRIO 4-band strict-priority mapping



Lower band number indicates higher transmission priority

Figure 2. DSCP-to-priority band mapping for PFIFO_Fast-Style and PRIO schedulers.

3.3 Queueing disciplines under test

Five queueing disciplines were compared. FIFO serves as the no-priority baseline. The PFIFO_FAST-style baseline was implemented as a three-band PRIO queue with FIFO children and a classifier that emulates a PFIFO_FAST-like DSCP mapping. The PRIO case extends this idea into a four-band strict-priority hierarchy so that control, status, telemetry, and lower-priority/default traffic can be separated more explicitly. FQ-PIE and FQ-CoDel represent fair-queue / active-queue-management approaches that attempt to isolate flows and suppress persistent queue growth.

These five cases were selected because together they represent three distinct design philosophies: pure best effort (FIFO), explicit class priority (PFIFO_FAST-style and PRIO), and fair queueing with AQM (FQ-PIE and FQ-CoDel). This makes it possible to study not only which scheduler performs best in this topology, but also why it performs that way.

3.4 Experimental scenarios

Three scenario families were executed. The scheduler-comparison scenario fixed the bottleneck at 10 Mbps, the number of followers at 4, the queue size at 40 packets, and the background rate at 12 Mbps per direction, and then compared all five schedulers with DSCP both disabled and enabled. The load-sweep scenario fixed the follower count at 4 and varied the background rate per direction across 8, 10, 12, 14, and 16 Mbps. The follower-sweep scenario fixed the offered background load at 12 Mbps per direction and varied the number of followers across 4, 8, and 16.

All scenario families were automated through shell scripts that passed arguments to the compiled ns-3 program. In total, the study produced 90 simulations: 10 scheduler-comparison runs, 50 load-sweep runs, and 30 follower-sweep runs. This structure is sufficient to expose both a single overloaded baseline point and two independent sensitivity studies.

Scenario family	Swept parameter	Fixed settings	DSCP modes	Runs
Scheduler comparison	Scheduler	Followers=4, bg=12 Mbps/dir, queue=40 pkts, bottleneck=10 Mbps, sim=20 s	Off and on	10
Load sweep	Background load	Followers=4, queue=40 pkts, bottleneck=10 Mbps, bg=8..16 Mbps/dir	Off and on	50
Follower sweep	Follower count	bg=12 Mbps/dir, queue=40 pkts, bottleneck=10 Mbps, followers=4/8/16	Off and on	30

3.5 Metrics and data processing

The simulator exports per-flow statistics to CSV, including transmitted and received packets, packet loss, throughput, mean delay, and mean jitter. These raw outputs are then aggregated into class-summary CSV files by grouping over scenario, scheduler, DSCP mode, follower count, background rate, and flow class. In addition to delay, loss, jitter, and throughput, the post-processing stage computes Jain fairness indices. Throughput fairness is computed directly from per-flow throughput values, while delay fairness is computed by applying Jain's index to inverse delay so that lower delay corresponds to better service. The final plots used in this report were generated automatically from these aggregated CSV files.

One limitation of the present dataset is that each configuration was executed with a single seed. That does not invalidate the trends, because many of the differences are extremely large, but it does mean that this report

should be interpreted as a controlled deterministic study rather than as a statistically replicated Monte Carlo analysis.

4. RESULTS

4.1 Baseline overloaded scheduler comparison

The baseline scheduler-comparison point uses 4 followers and 12 Mbps of background traffic per direction over a 10 Mbps bottleneck. This is the clearest view of scheduler behavior under overload. The control-plane results show an immediate separation between the schedulers. FIFO produces 40.8 ms mean control delay, 61.4% control packet loss, 1.0 ms mean jitter, and only 0.227 Mbps control throughput. FQ-PIE and FQ-CoDel reduce control delay to 8.9 ms, eliminate control loss, hold jitter at 0.2 ms, and preserve 0.589 Mbps control throughput even without DSCP. PFIFO_FAST-style and PRIO behave like FIFO when DSCP is disabled, but once DSCP is enabled they match the strong low-delay, zero-loss control behavior of the fair-queue schedulers.

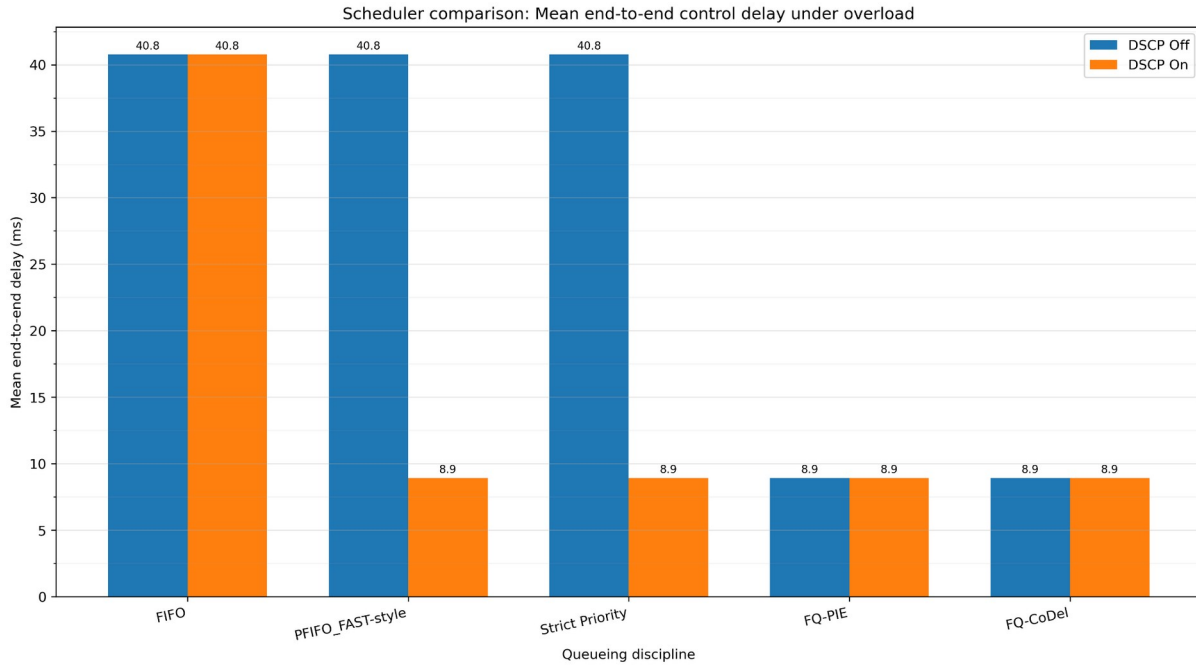


Figure 3. Mean end-to-end control delay under the baseline overloaded comparison point.

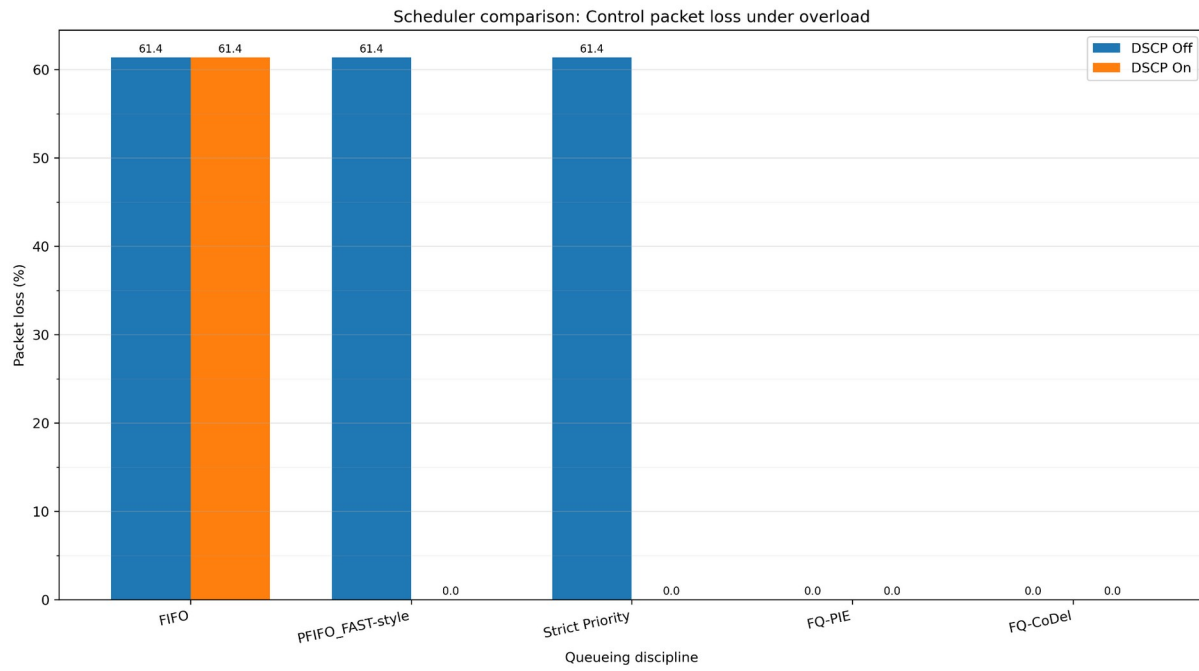


Figure 4. Control packet loss under the baseline overloaded comparison point.

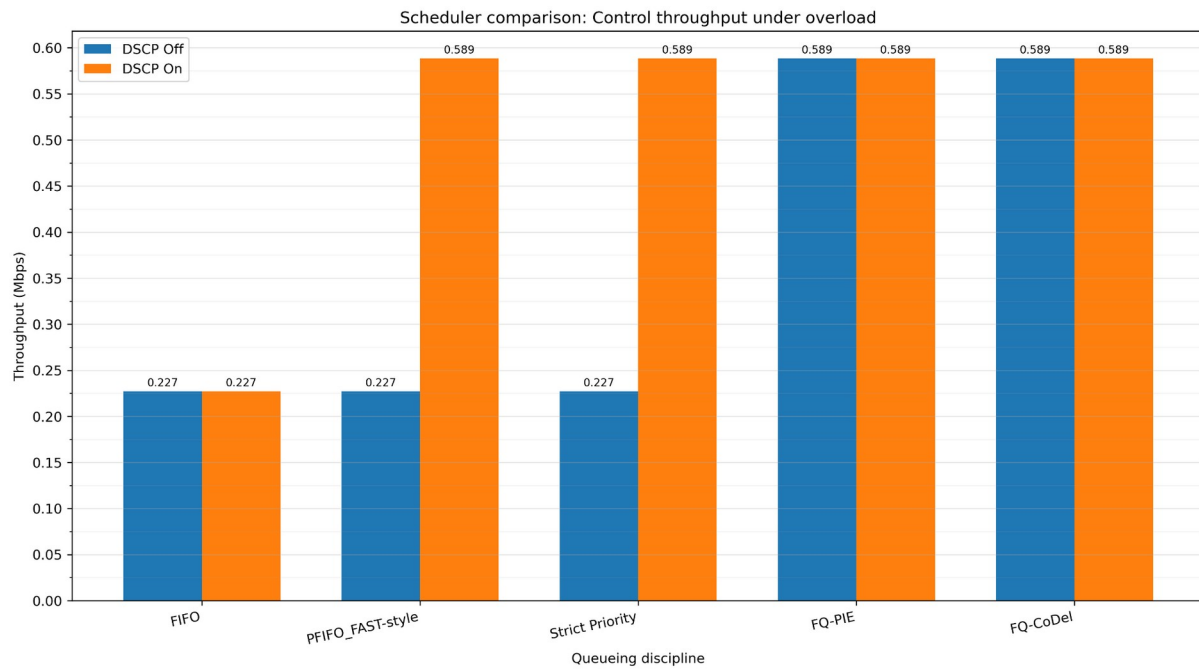


Figure 5. Control throughput under the baseline overloaded comparison point.

Scheduler	DSCP	Delay (ms)	Loss (%)	Jitter (ms)	Throughput (Mbps)
FIFO	Off	40.8	61.4	1.0	0.227
FIFO	On	40.8	61.4	1.0	0.227
PFIFO_FAST-style	Off	40.8	61.4	1.0	0.227
PFIFO_FAST-style	On	8.9	0.0	0.2	0.589

Strict Priority	Off	40.8	61.4	1.0	0.227
Strict Priority	On	8.9	0.0	0.2	0.589
FQ-PIE	Off	8.9	0.0	0.2	0.589
FQ-PIE	On	8.9	0.0	0.2	0.589
FQ-CoDel	Off	8.9	0.0	0.2	0.589
FQ-CoDel	On	8.9	0.0	0.2	0.589

Table 1. Baseline overloaded control-plane results for the scheduler comparison scenario.

The DSCP-specific comparison for priority queueing makes the key result especially clear. For both PFIFO_FAST-style and strict-priority PRIO, enabling DSCP reduces mean control delay by 31.9 ms and reduces control loss by 61.4 percentage points. In other words, explicit priority classification is effective in this topology, but only when the packets are actually marked so that the classifier can distinguish them.

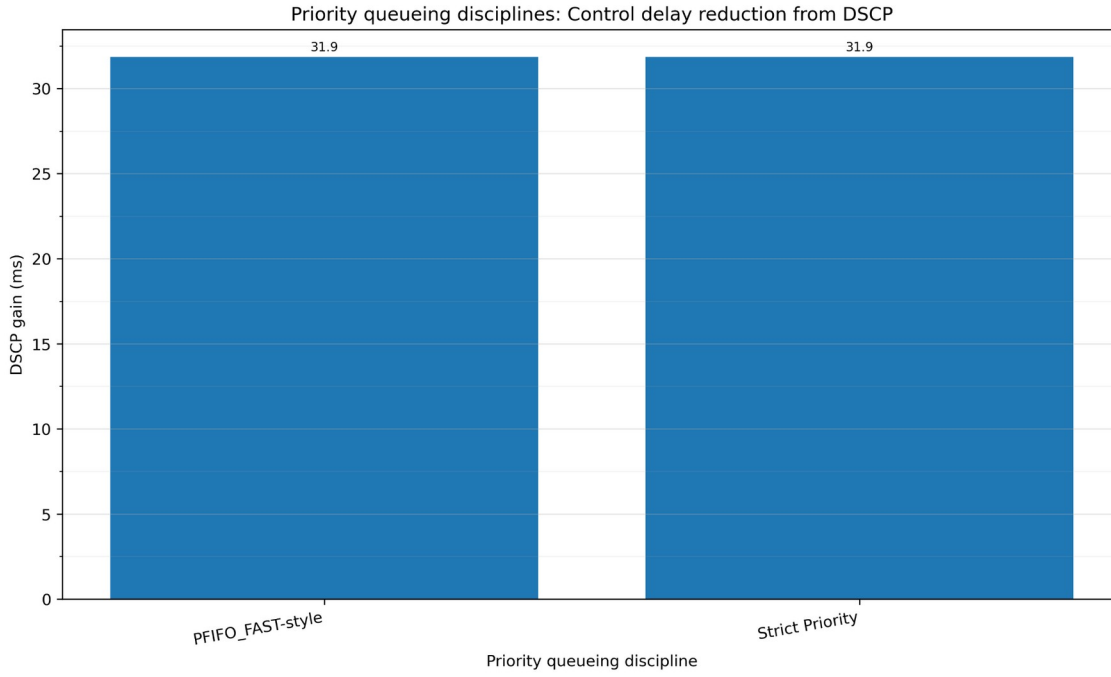


Figure 6. Control delay reduction achieved by DSCP when priority schedulers are used.

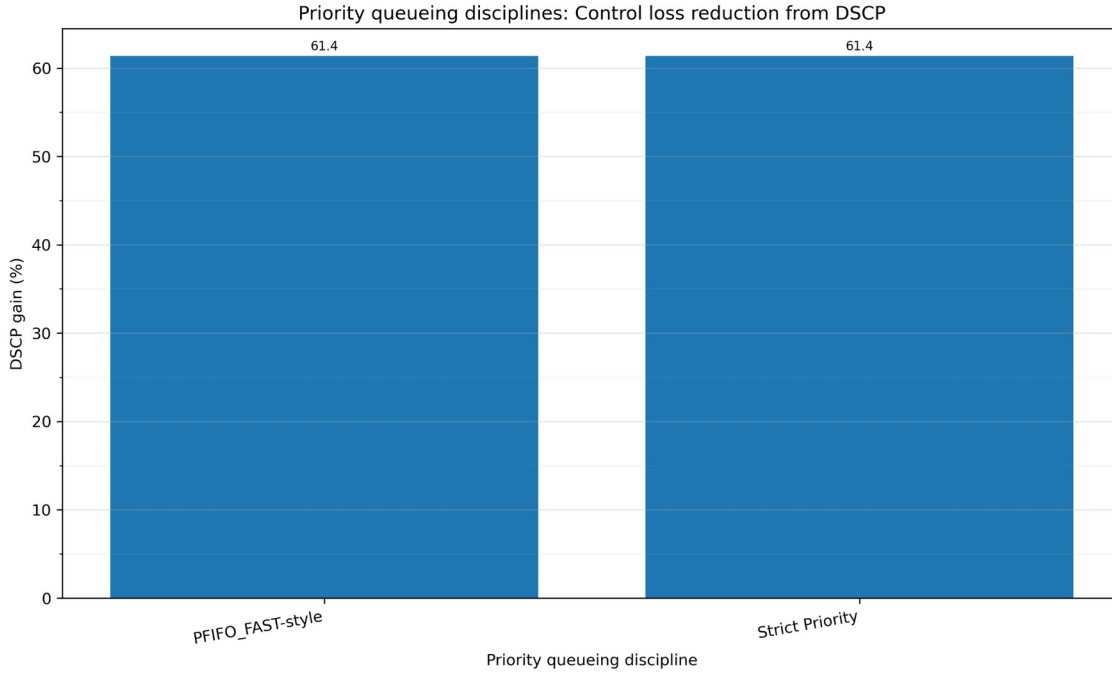


Figure 7. Control loss reduction achieved by DSCP when priority schedulers are used.

4.2 Per-class behavior and trade-offs

Control traffic is not the only class of interest. The per-class plots reveal how each scheduler redistributes service across the system. With DSCP enabled, the fair-queue schedulers keep control, status, telemetry, and bulk delays low while still allowing background throughput to remain high. The strict-priority schedulers, on the other hand, protect the higher-priority classes by pushing more queueing burden onto lower-priority traffic. This is not inherently wrong; it is in fact the intended behavior of priority scheduling. However, it means that the choice between fair queueing and strict priority should depend on whether the deployment values low-latency isolation above overall balance.

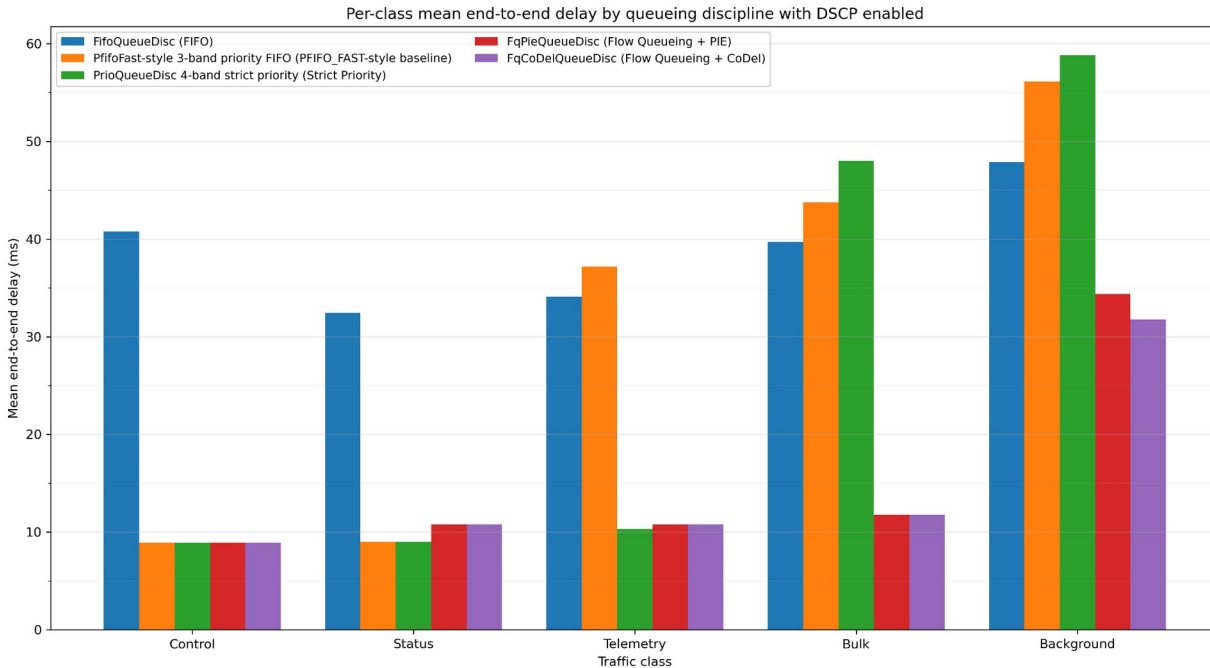


Figure 8. Per-class delay by scheduler with DSCP enabled.

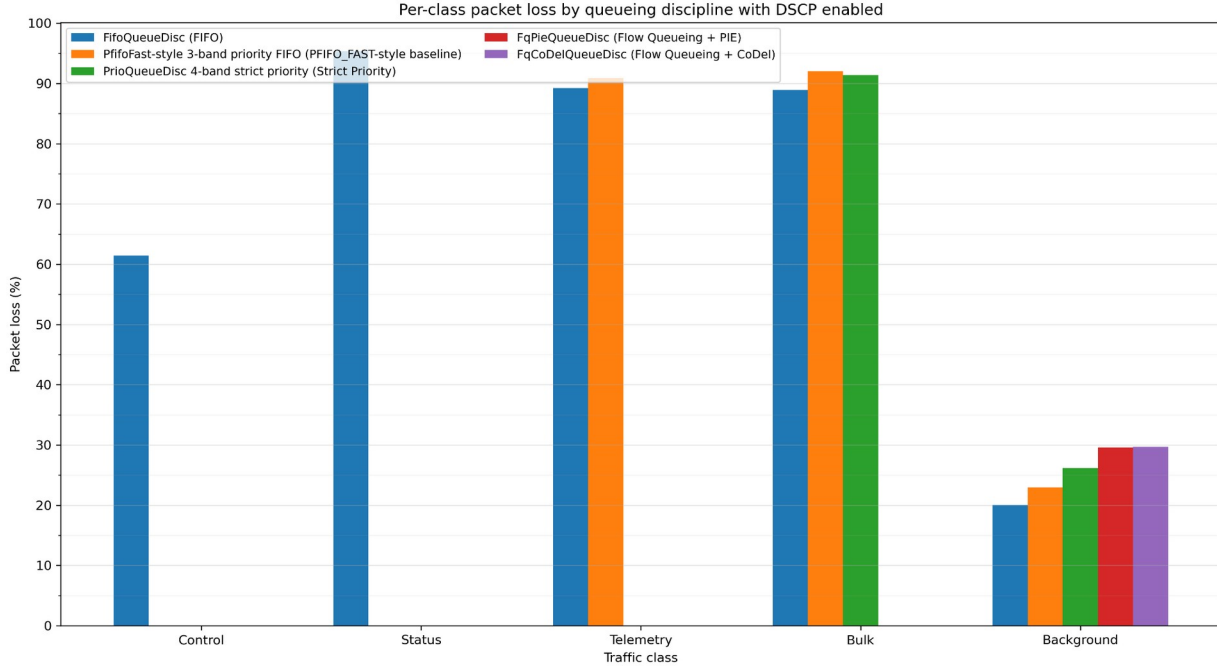


Figure 9. Per-class packet loss by scheduler with DSCP enabled.

Heatmaps provide a compact cross-class summary of the same DSCP-enabled comparison point. They make it easy to see, in one view, how FIFO leaves control and reverse-direction application traffic exposed to high delay and loss, while the priority and fair-queue cases concentrate better service on the latency-sensitive classes. They are included here as a dense visual summary rather than as the primary explanatory figures.

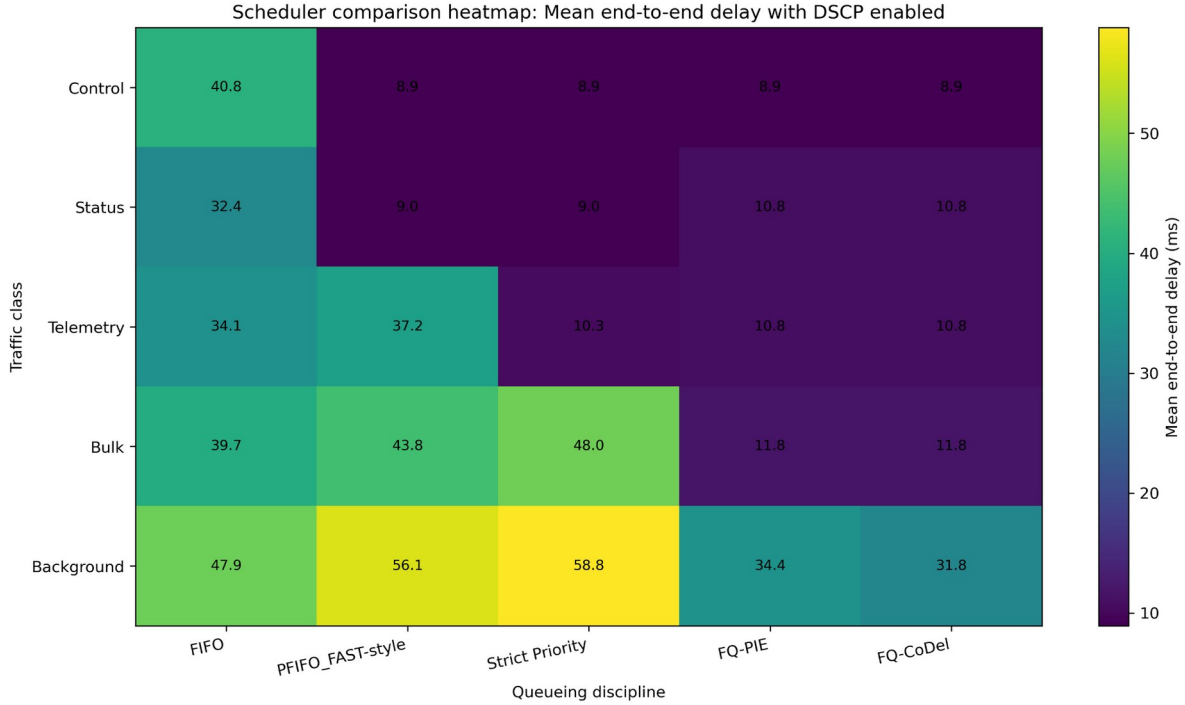


Figure 10. Heatmap summary of per-class mean end-to-end delay with DSCP enabled.

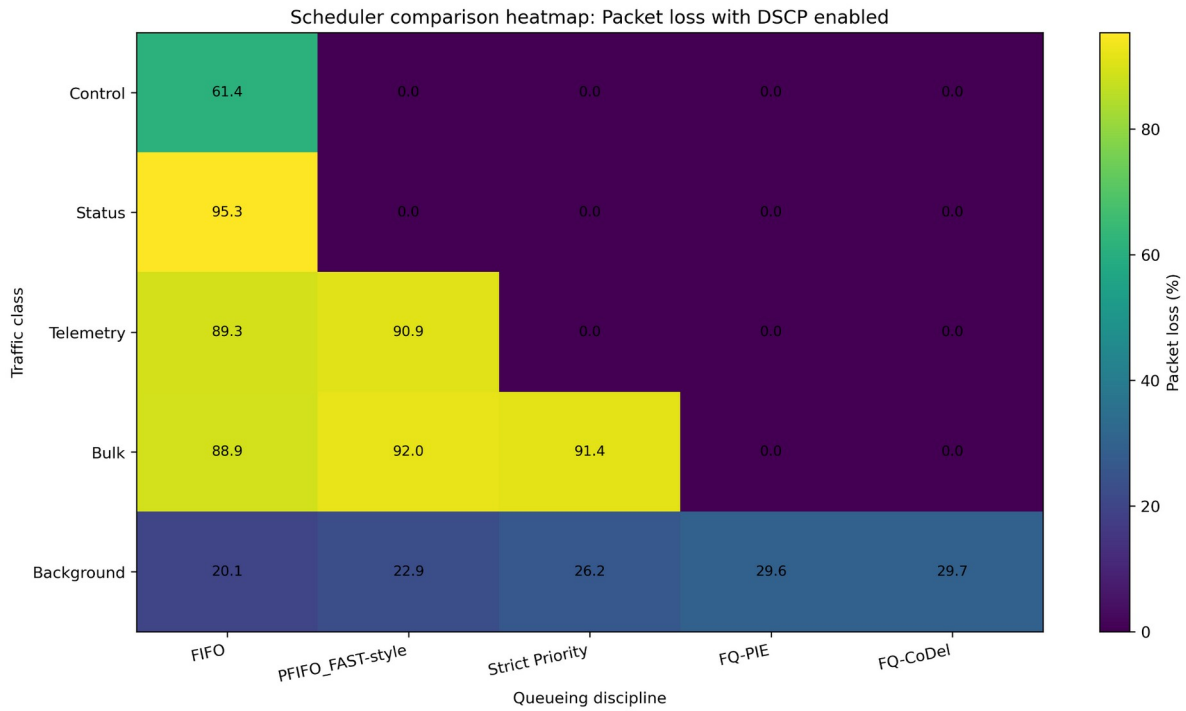


Figure 11. Heatmap summary of per-class packet loss with DSCP enabled.

A noteworthy detail appears in telemetry performance. Under strict-priority PRIO with DSCP enabled, telemetry delay falls from 34.1 ms to 10.3 ms and telemetry loss falls from 89.3% to 0.0%. By contrast, the PFIFO_FAST-style configuration does not improve telemetry at the same operating point; telemetry delay slightly worsens and telemetry loss remains high. This happens because the PFIFO_FAST-style three-band arrangement separates control and status but still leaves telemetry grouped with lower-priority/default traffic. The four-band PRIO design therefore provides a more complete QoS hierarchy for this workload.

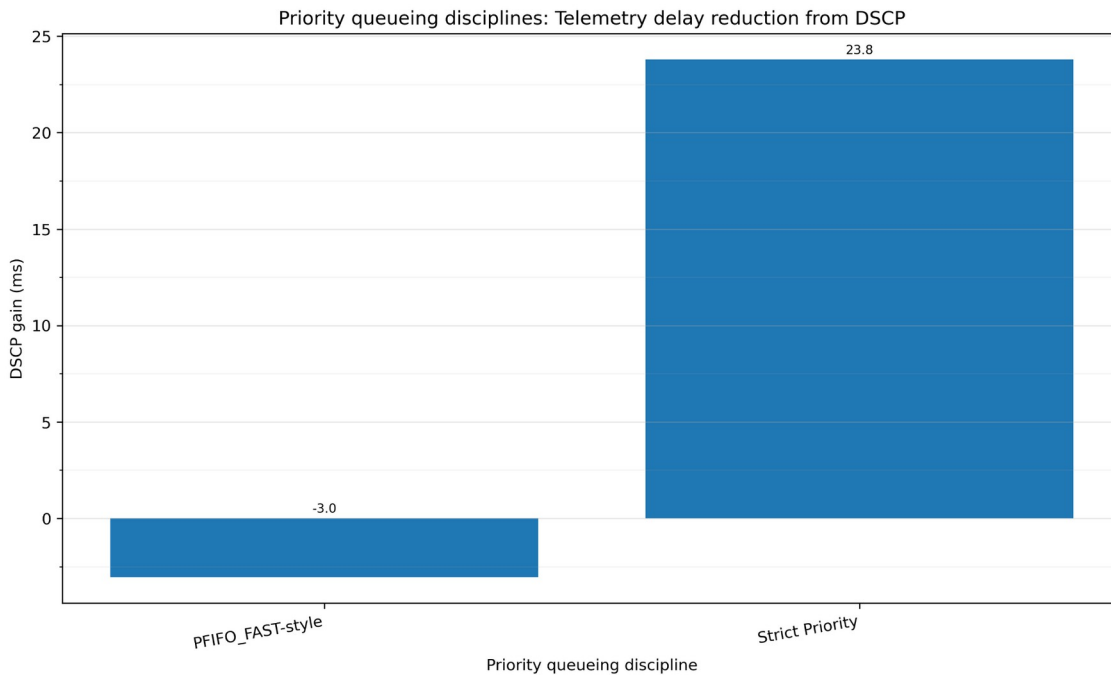


Figure 12. Telemetry delay reduction from DSCP for the priority schedulers.

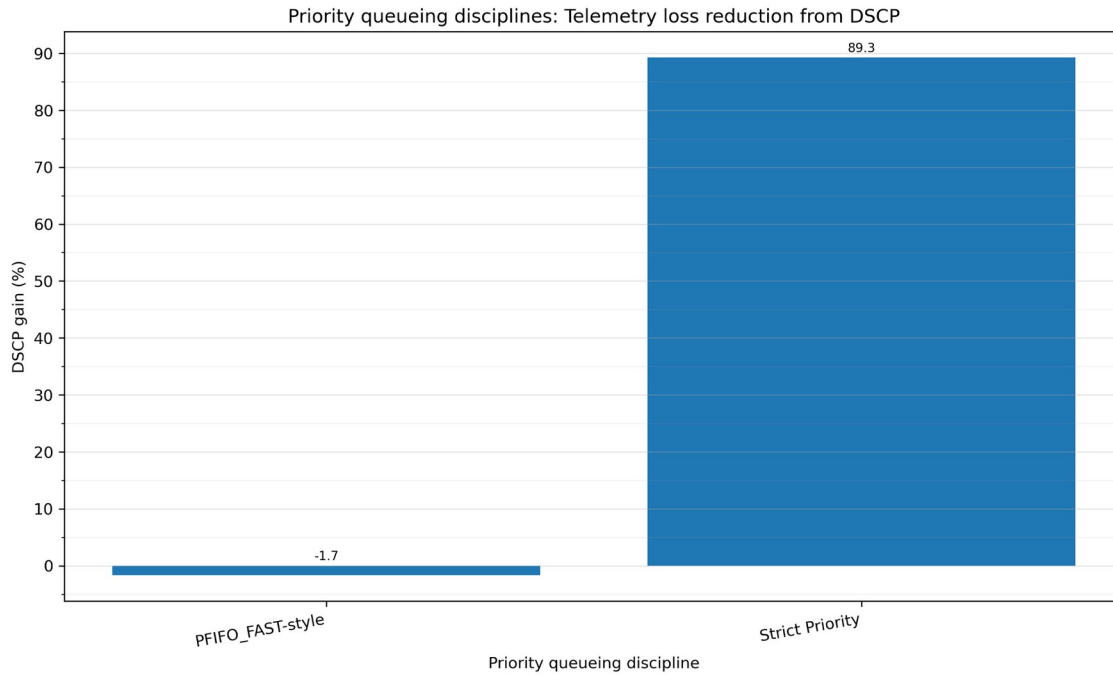


Figure 13. Telemetry loss reduction from DSCP for the priority schedulers.

4.3 Load sweep

The load sweep tests whether the scheduler ranking observed at the baseline point remains stable as background traffic is increased. The answer is yes. At 8 Mbps of background load per direction, all schedulers behave similarly and control traffic sees low delay and zero loss. Once the offered background load reaches 10 Mbps and then exceeds the 10 Mbps bottleneck, FIFO begins to degrade rapidly. Without DSCP, the PFIFO_FAST-style and PRIO schedulers follow the same poor trajectory because they are not yet separating classes. FQ-PIE and FQ-CoDel remain essentially flat across the full sweep, keeping control delay near 8.9 ms and control loss at zero. When DSCP is enabled, PFIFO_FAST-style and PRIO also flatten, indicating that the classification mechanism is the missing ingredient rather than the queue structure alone.

Load sweep: Mean end-to-end control delay vs offered background load

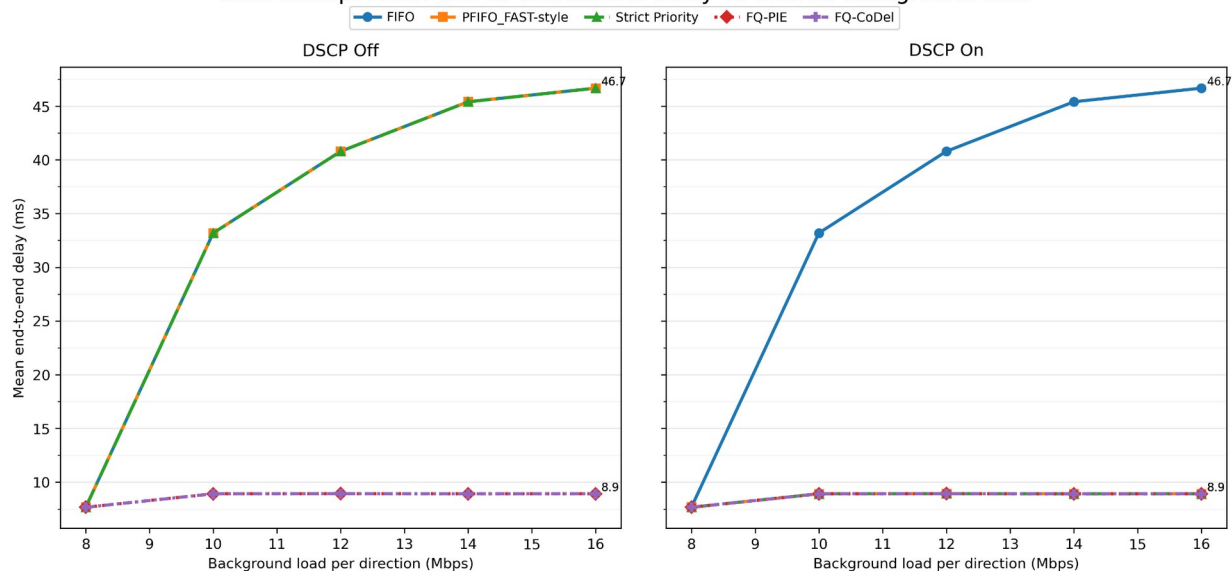


Figure 14. Mean control delay across the offered-load sweep.

Load sweep: Control packet loss vs offered background load

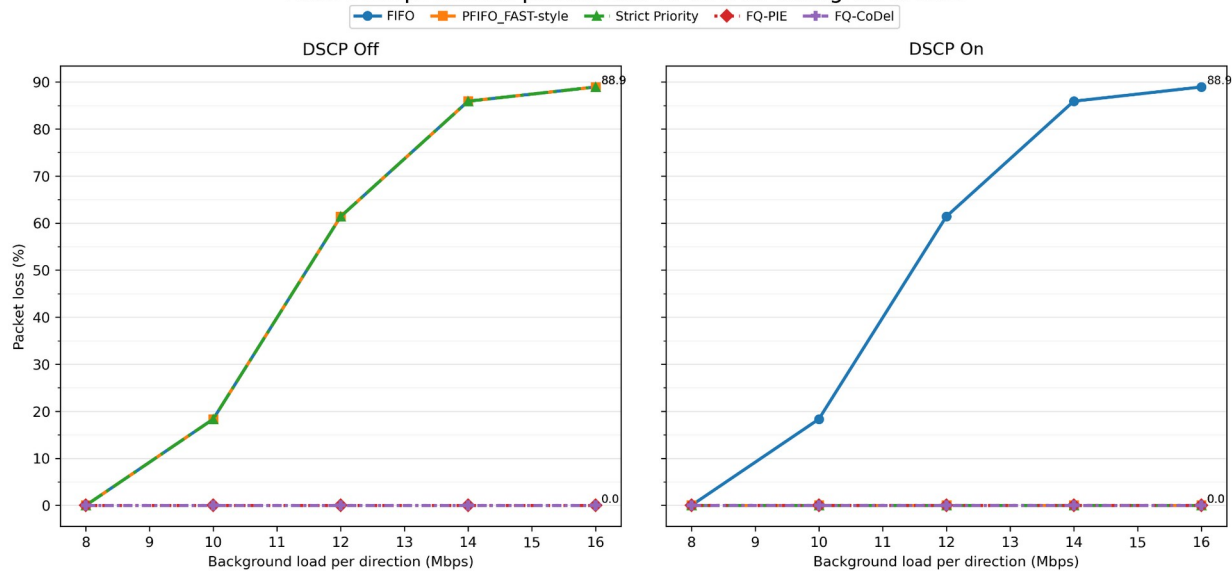


Figure 15. Control loss across the offered-load sweep.

Load sweep: Control throughput vs offered background load

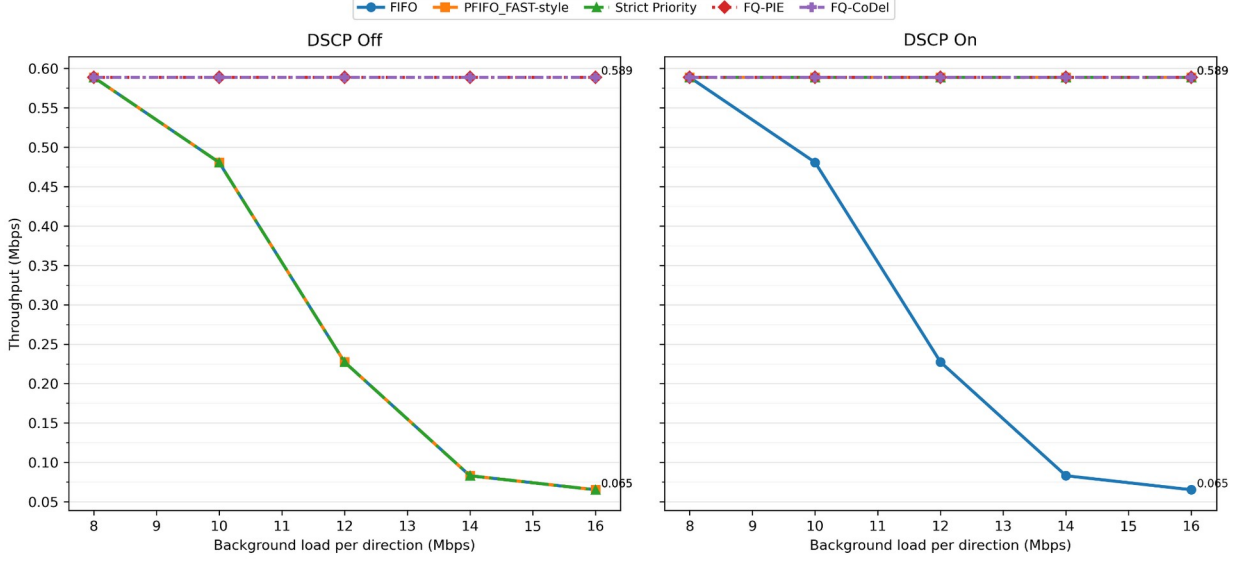


Figure 16. Control throughput across the offered-load sweep.

4.4 Follower sweep

The follower sweep studies scaling pressure from the application side rather than the background-load side. As the number of followers increases from 4 to 8 and then 16, the same broad ranking remains visible. FIFO control loss rises from 61.4% to 78.6% and then 85.9%, while FQ-PIE and FQ-CoDel keep control loss at zero throughout. With DSCP enabled, the PFIFO_FAST-style and PRIO schedulers track the fair-queue schedulers rather than FIFO, again showing that class marking is decisive. Control delay under FIFO declines somewhat as the number of followers rises, but this does not indicate improvement; it occurs together with very high packet loss and collapsing service, so many packets are simply not surviving long enough to contribute to a larger delay average.

Follower sweep: Mean end-to-end control delay vs number of followers

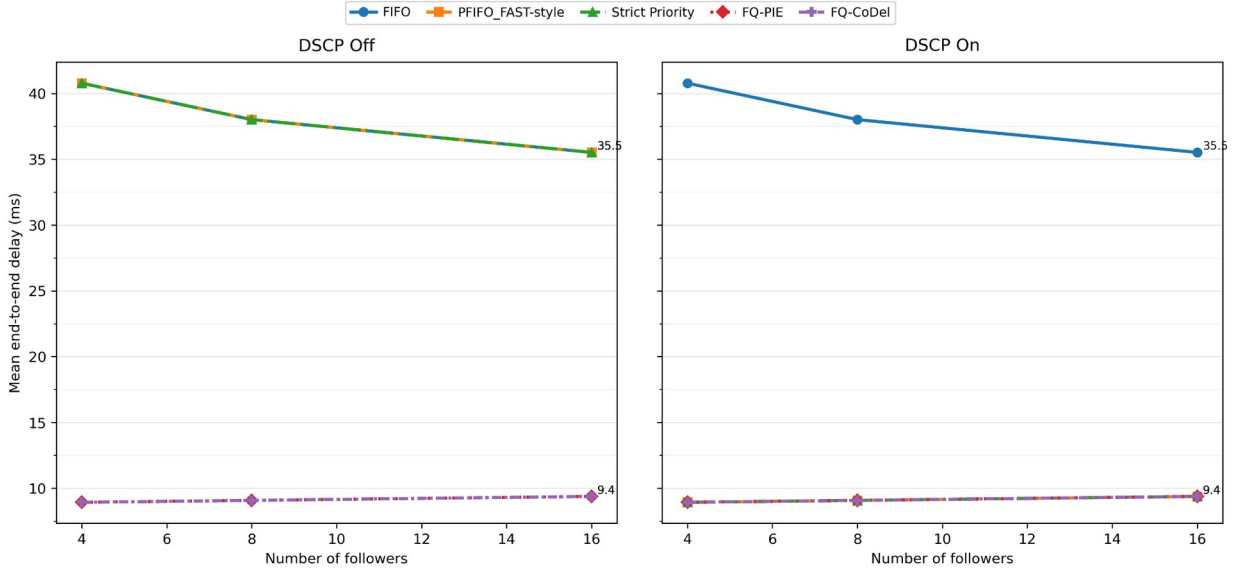


Figure 17. Mean control delay across the follower-count sweep.

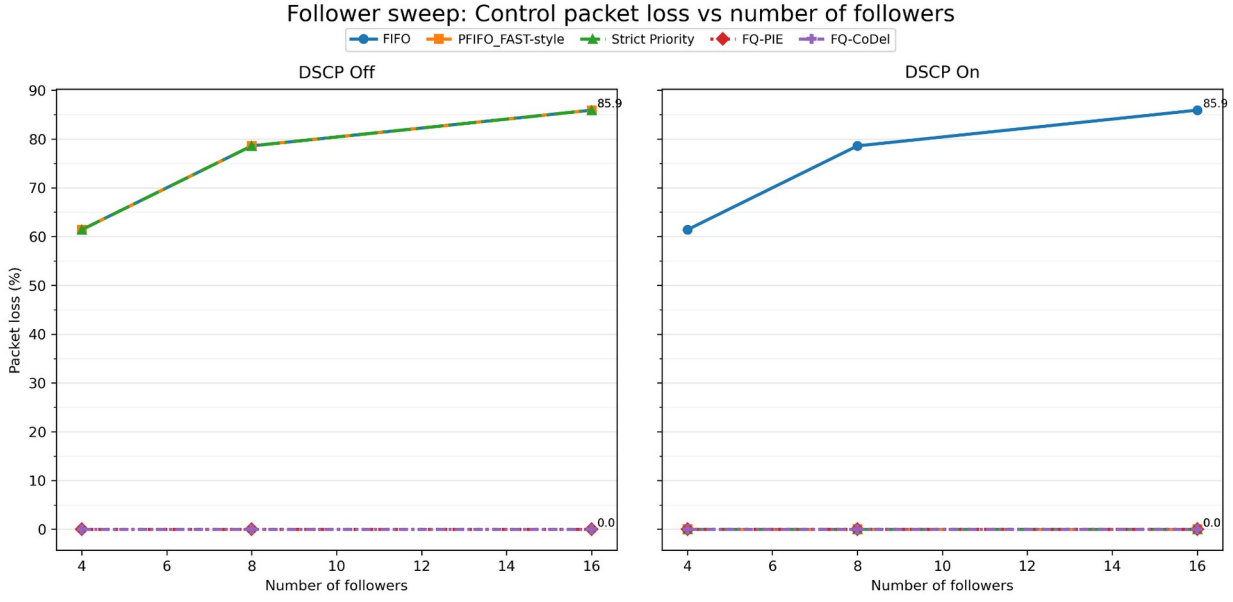


Figure 18. Control loss across the follower-count sweep.

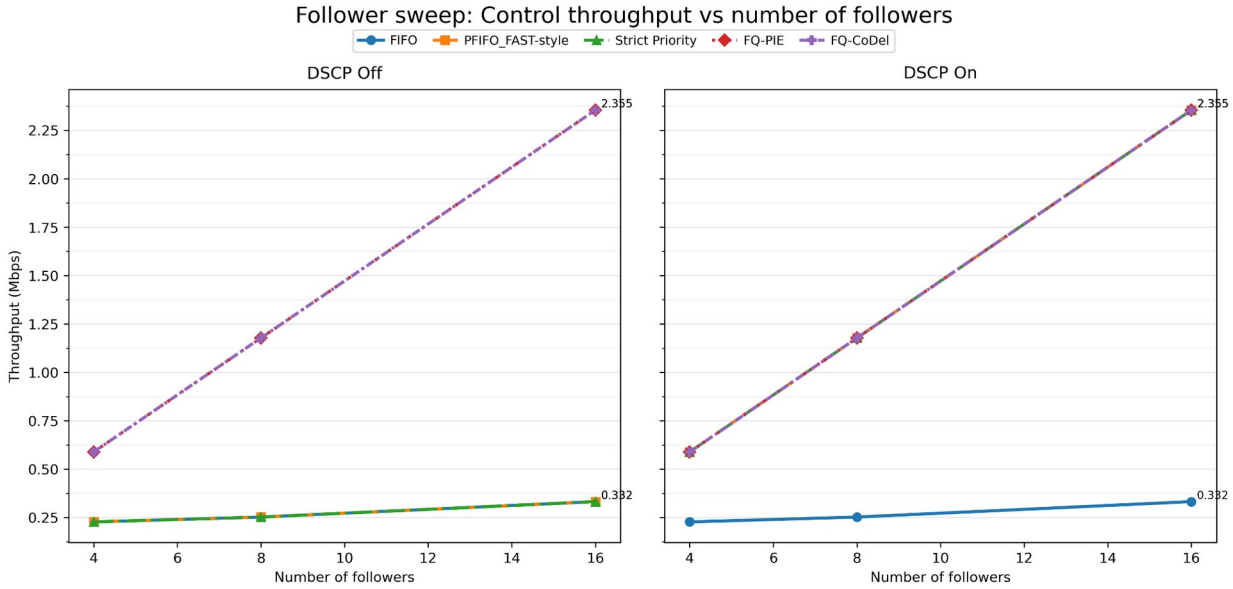


Figure 19. Control throughput across the follower-count sweep.

4.5 Fairness analysis

Fairness was evaluated using Jain indices computed per application class. The fairness plots complement the scheduler comparison by showing that strong control protection can coexist with very different service-distribution patterns. In the scheduler-comparison scenario with DSCP enabled, throughput fairness for control becomes perfect for PFIFO_FAST-style, PRIO, FQ-PIE, and FQ-CoDel, while FIFO remains lower. Delay fairness is generally high across all schedulers, but some degradation appears in lower-priority classes under overloaded conditions. The load and follower sweeps show that FIFO fairness collapses as overload grows, especially for telemetry and bulk throughput, whereas the fair-queue schedulers maintain near-perfect fairness for the application classes they protect.

Scheduler comparison: Jain throughput fairness index with DSCP enabled

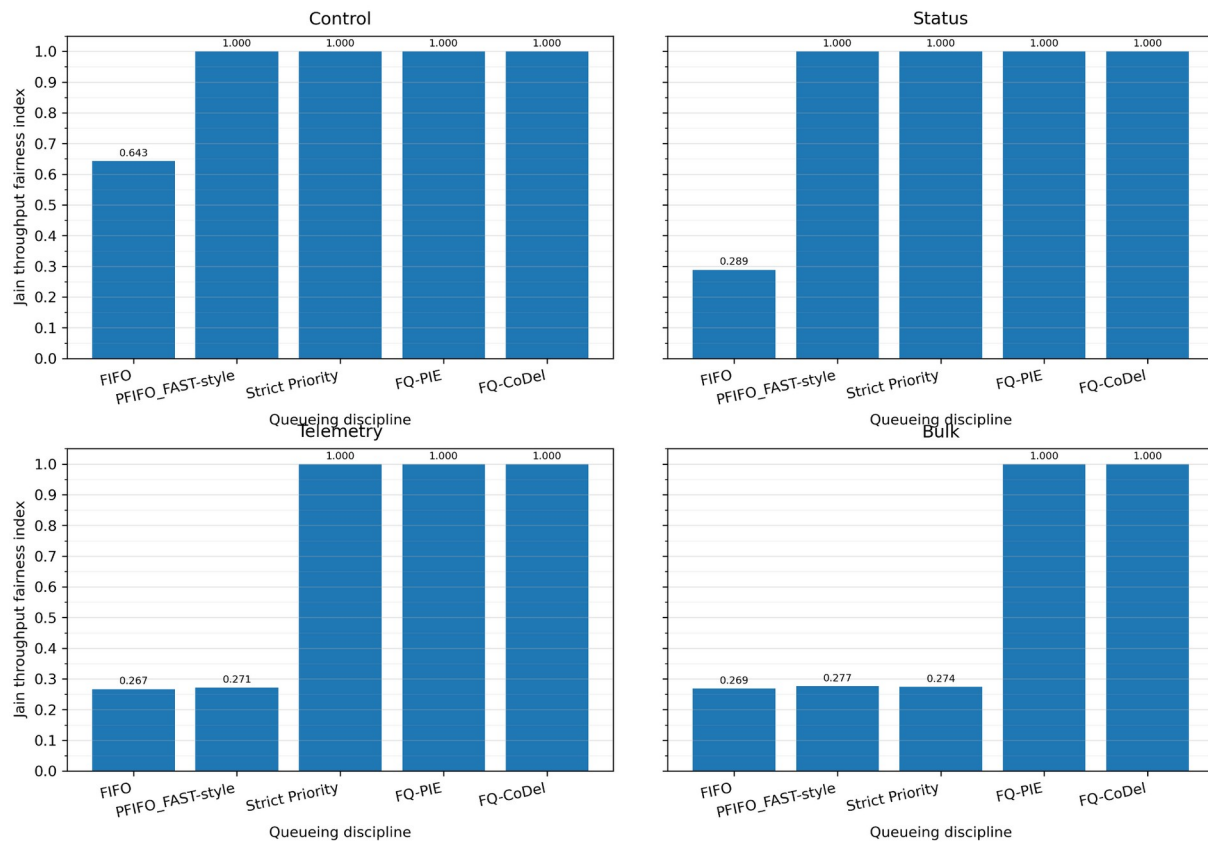


Figure 20. Jain throughput fairness index for the scheduler-comparison scenario with DSCP enabled.

Load sweep: Jain throughput fairness index with DSCP enabled

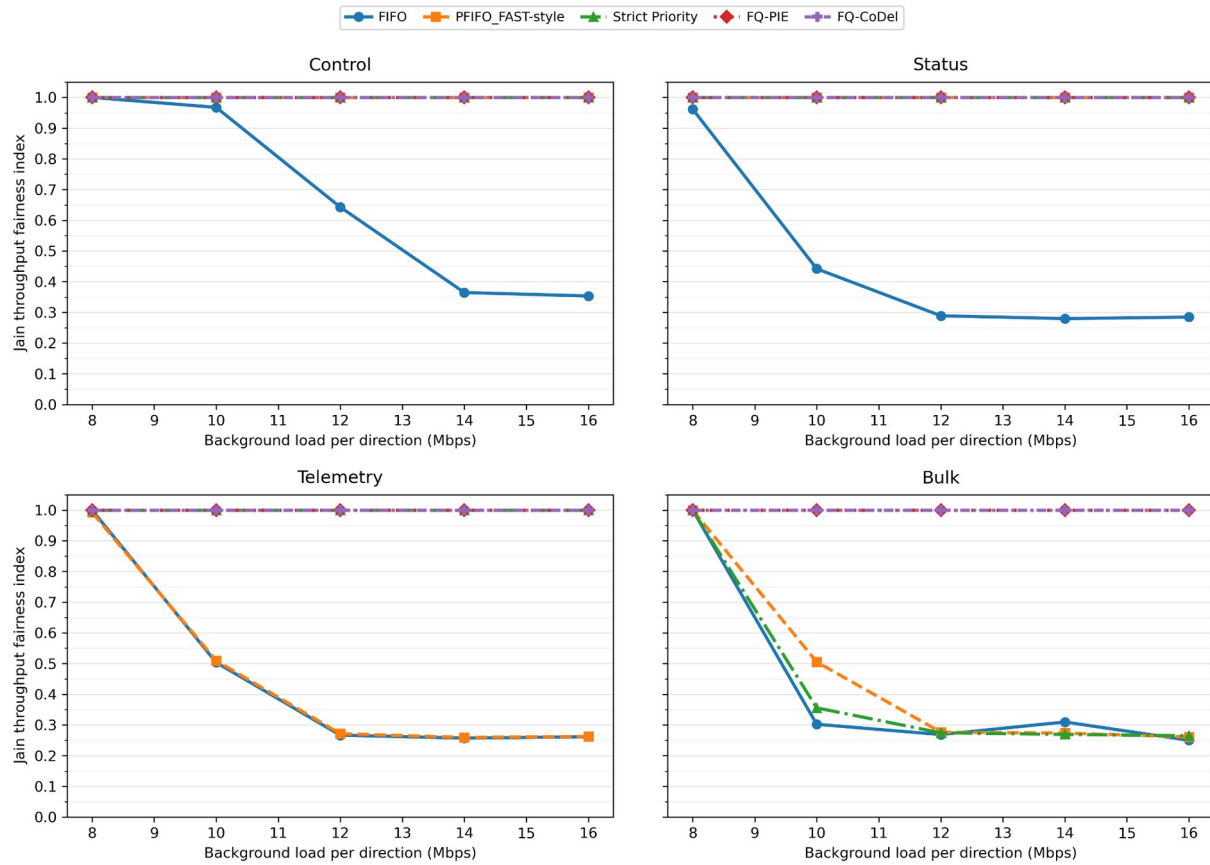


Figure 21. Jain throughput fairness index across the offered-load sweep with DSCP enabled.

Follower sweep: Jain throughput fairness index with DSCP enabled

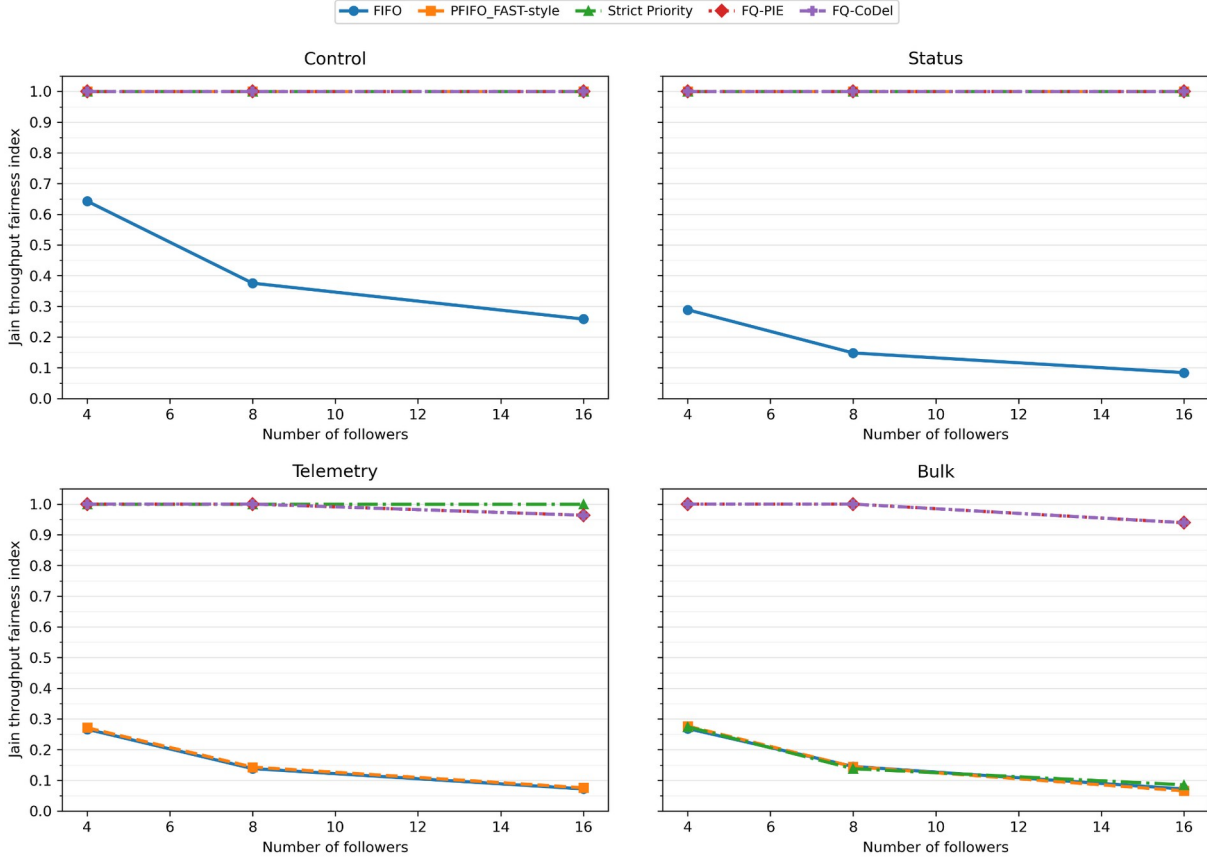


Figure 22. Jain throughput fairness index across the follower-count sweep with DSCP enabled.

5. DISCUSSION AND CONCLUSIONS

This project demonstrates that QoS enforcement for bidirectional UDP control traffic is fundamentally a queue-management problem. When all packets share the bottleneck in a best-effort manner, control performance can become unacceptable even in a simple topology. The baseline overloaded case showed that FIFO allowed control delay to rise to 40.8 ms and control loss to rise to 61.4%. Such a result would be difficult to tolerate in a real control loop.

The non-trivial aspect of the project was not only building the ns-3 topology, but designing a workload in which multiple application classes simultaneously contend across both directions of a bottleneck while remaining interpretable enough to analyze. The study had to distinguish control from reverse-direction status and telemetry traffic, create overload in a controlled way, automate multiple scenario families, aggregate per-flow outputs into class-level summaries, and then compare schedulers using both direct QoS metrics and fairness metrics.

The strongest practical conclusion is that two different solution families work well in this setting. FQ-PIE and FQ-CoDel provide robust control protection even without DSCP-specific tuning in the presented topology. PFIFO_FAST-style and PRIO can also achieve very strong control behavior, but only when DSCP marking is enabled and the classifier has enough bands to separate the classes that matter. Among the priority approaches, PRIO is the more expressive design because it can isolate telemetry more effectively than the simpler PFIFO_FAST-style mapping.

Alternative approaches could extend the study in several directions. First, future work should add repeated runs with multiple seeds so that confidence intervals can be reported. Second, the queue size, access-link capacity, and background burstiness could be swept to determine how sensitive the ranking is to short-term queue transients. Third, the model could be expanded to include wireless links, mixed TCP/UDP cross traffic, or application-level deadlines to move closer to a production deployment. Finally, a more explicit weighted scheduler could be added if a future ns-3 configuration is desired that more directly resembles weighted fair queueing.

In summary, the report finds that packet scheduling can successfully enforce QoS for bidirectional UDP-based control and telemetry traffic, but the outcome depends strongly on whether the scheduler can actually distinguish classes and prevent latency-sensitive traffic from being trapped in a common backlog. For the topology and workloads studied here, FQ-PIE, FQ-CoDel, and DSCP-enabled PRIO were the most effective configurations.

6. REFERENCES

- [1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Pearson, 2021.
- [2] J. Postel, "User Datagram Protocol," RFC 768, Aug. 1980.
- [3] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474, Dec. 1998.
- [4] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: an overview," RFC 1633, June 1994.
- [5] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1-12, Sept. 1989.
- [6] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365-386, Aug. 1995.
- [7] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344-357, June 1993.
- [8] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375-385, June 1996.
- [9] K. Nichols and V. Jacobson, "Controlling queue delay," *ACM Queue*, vol. 10, no. 5, May 2012.
- [10] A. Aijaz et al., "QoS Provisioning for Industrial IoT: Evaluation of Packet Scheduling in Mixed Real-Time and Best-Effort Traffic," *IEEE Access*, vol. 7, pp. 89542-89556, 2019.
- [11] R. Jain, D.-M. Chiu, and W. R. Hawe, "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems," *DEC Research Report TR-301*, 1984.
- [12] T. R. Henderson et al., "The ns-3 Network Simulator," in *Modeling and Tools for Network Simulation*. Berlin, Heidelberg: Springer, 2008, pp. 15-34.

APPENDIX A. CODE LISTING

Listing A.1. Commented ns-3 source used to generate the study results.

```
#include "ns3/applications-module.h"
#include "ns3/core-module.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/ipv4-flow-classifier.h"
#include "ns3/ipv4-queue-disc-item.h"
#include "ns3/ipv4-packet-filter.h"
#include "ns3/internet-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/traffic-control-module.h"

#include <fstream>
#include <iomanip>
#include <sstream>
#include <string>
#include <vector>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("QosUdpStudy");

// Small study driver for comparing a few queueing disciplines under the same UDP workload.
// Nothing too fancy here, just trying to keep the topology and the CSV output predictable.

class DscpBandIpv4PacketFilter : public Ipv4PacketFilter
{
public:
    static TypeId GetTypeId()
    {
        static TypeId tid =
            TypeId("ns3::DscpBandIpv4PacketFilter")
                .SetParent<Ipv4PacketFilter>()
                .SetGroupName("TrafficControl")
                .AddConstructor<DscpBandIpv4PacketFilter>();
        return tid;
    }

private:
    bool CheckProtocol(Ptr<QueueDiscItem> item) const override
    {
        return DynamicCast<Ipv4QueueDiscItem>(item) != nullptr;
    }

    int32_t DoClassify(Ptr<QueueDiscItem> item) const override
    {
        Ptr<Ipv4QueueDiscItem> ipv4Item = DynamicCast<Ipv4QueueDiscItem>(item);
        if (!ipv4Item)
        {
            return PacketFilter::PF_NO_MATCH;
        }

        uint8_t dscp = ipv4Item->GetHeader().GetTos() >> 2;

        // This one is just the straightforward 4-band mapping I used for the strict-priority case.
        // Kept it explicit on purpose so it is easy to sanity-check later.
        // Explicit DSCP -> band mapping
        // band 0 = highest priority
    }
};
```

```
// control: AF42 (36)
// status: AF43 (38)
// telemetry: AF41 (34)
// bulk/background: CS1/default
if (dscp == 36)
{
    return 0;
}
if (dscp == 38)
{
    return 1;
}
if (dscp == 34)
{
    return 2;
}
return 3;
}
};
```

```
NS_OBJECT_ENSURE_REGISTERED(DscpBandIpv4PacketFilter);
```

```
class DscpBand3Ipv4PacketFilter : public Ipv4PacketFilter
```

```
{
public:
    static TypeId GetTypeId()
    {
        static TypeId tid =
            TypeId("ns3::DscpBand3Ipv4PacketFilter")
                .SetParent<Ipv4PacketFilter>()
                .SetGroupName("TrafficControl")
                .AddConstructor<DscpBand3Ipv4PacketFilter>();
        return tid;
    }
}
```

```
private:
```

```
bool CheckProtocol(Ptr<QueueDiscItem> item) const override
{
    return DynamicCast<Ipv4QueueDiscItem>(item) != nullptr;
}
```

```
int32_t DoClassify(Ptr<QueueDiscItem> item) const override
```

```
{
    Ptr<Ipv4QueueDiscItem> ipv4Item = DynamicCast<Ipv4QueueDiscItem>(item);
    if (!ipv4Item)
    {
        return PacketFilter::PF_NO_MATCH;
    }
}
```

```
uint8_t dscp = ipv4Item->GetHeader().GetTos() >> 2;
```

```
// Same idea as above, but trimmed down to 3 bands to mimic pfifo_fast-style behavior.
```

```
// Not a perfect clone, but close enough for the comparison I wanted to run.
```

```
// 3-band mapping for pfifo_fast-style behavior
```

```
// AF42 -> band 0 (highest)
```

```
// AF43 -> band 1
```

```
// AF41 -> band 2
```

```
// everything else -> band 2
```

```
if (dscp == 36) // AF42
```

```
{
    return 0;
}
```

```
if (dscp == 38) // AF43
```

```
{
    return 1;
}
return 2;    // AF41, CS1, background, default
}
};

NS_OBJECT_ENSURE_REGISTERED(DscpBand3Ipv4PacketFilter);

struct ClassSpec
{
    std::string name;
    uint16_t portBase;
    uint32_t packetBytes;
    double ratePps;
    uint8_t dscpDecimal;
    bool downlink;
};

struct RunConfig
{
    std::string runId = "run001";
    std::string scenario = "baseline";
    std::string scheduler = "fifo";
    bool dscp = false;
    uint32_t followers = 4;
    uint32_t queuePkts = 40;

    double bottleneckMbps = 10.0;
    double accessMbps = 100.0;
    double simTime = 20.0;

    double bgRateMbps = 6.0;
    bool burstyBg = true;
};

static uint8_t
MakeTos(bool dscpEnabled, uint8_t dscpDecimal)
{
    // If DSCP is off, just leave the TOS byte alone and let everything blend together.
    if (!dscpEnabled)
    {
        return 0;
    }
    return static_cast<uint8_t>(dscpDecimal << 2);
}

static std::string
SchedulerLabel(const std::string& key)
{
    if (key == "fifo") return "FifoQueueDisc (FIFO)";
    if (key == "pfifo_fast") return "PfifoFast-style 3-band priority FIFO (RR-family baseline)";
    if (key == "fq_pie") return "FqPieQueueDisc (WFQ-family analog)";
    if (key == "prio") return "PrioQueueDisc 4-band strict priority (PRIO)";
    if (key == "fq_codel") return "FqCoDelQueueDisc (CoDel-family fair queue)";
    return key;
}

static void
InstallRootQdiscOnDevice(Ptr<NetDevice> dev, const std::string& scheduler, uint32_t queuePkts)
{
    TrafficControlHelper tch;
    std::ostringstream maxSize;
    maxSize << queuePkts << "p";
```

```
tch.Uninstall(dev);

if (scheduler == "fifo")
{
    tch.SetRootQueueDisc("ns3::FifoQueueDisc", "MaxSize", StringValue(maxSize.str()));
    tch.Install(dev);
}
else if (scheduler == "pfifo_fast")
{
    // Stable pfifo_fast-style implementation:
    // 3-band priority FIFO with explicit DSCP-based packet filter.
    uint16_t handle = tch.SetRootQueueDisc("ns3::PrioQueueDisc",
        "Priomap",
        StringValue("0 1 2 2 0 1 2 2 0 1 2 2 0 1 2 2"));

    TrafficControlHelper::ClassIdList cids =
        tch.AddQueueDiscClasses(handle, 3, "ns3::QueueDiscClass");

    for (uint32_t i = 0; i < 3; ++i)
    {
        tch.AddChildQueueDisc(handle, cids[i], "ns3::FifoQueueDisc",
            "MaxSize", StringValue(maxSize.str()));
    }

    tch.AddPacketFilter(handle, "ns3::DscpBand3Ipv4PacketFilter");
    tch.Install(dev);
}
else if (scheduler == "fq_codel")
{
    tch.SetRootQueueDisc("ns3::FqCoDelQueueDisc", "MaxSize", StringValue(maxSize.str()));
    tch.Install(dev);
}
else if (scheduler == "fq_pie")
{
    tch.SetRootQueueDisc("ns3::FqPieQueueDisc", "MaxSize", StringValue(maxSize.str()));
    tch.Install(dev);
}
else if (scheduler == "prio")
{
    // Strict 4-band priority with explicit DSCP-based packet filter.
    uint16_t handle = tch.SetRootQueueDisc("ns3::PrioQueueDisc",
        "Priomap",
        StringValue("0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3"));

    TrafficControlHelper::ClassIdList cids =
        tch.AddQueueDiscClasses(handle, 4, "ns3::QueueDiscClass");

    for (uint32_t i = 0; i < 4; ++i)
    {
        tch.AddChildQueueDisc(handle, cids[i], "ns3::FifoQueueDisc",
            "MaxSize", StringValue(maxSize.str()));
    }

    tch.AddPacketFilter(handle, "ns3::DscpBandIpv4PacketFilter");
    tch.Install(dev);
}
else
{
    NS_FATAL_ERROR("Unsupported scheduler: " << scheduler);
}
}
```

```
int
main(int argc, char* argv[])
{
    RunConfig cfg;

    CommandLine cmd(_FILE_);
    cmd.AddValue("runId", "Run identifier", cfg.runId);
    cmd.AddValue("scenario", "Scenario name", cfg.scenario);
    cmd.AddValue("scheduler", "fifo|pfifo_fast|fq_pie|prio|fq_codel", cfg.scheduler);
    cmd.AddValue("dscp", "Enable DSCP marking in IP header", cfg.dscp);
    cmd.AddValue("followers", "Number of followers", cfg.followers);
    cmd.AddValue("queuePkts", "Queue size in packets", cfg.queuePkts);
    cmd.AddValue("bottleneckMbps", "Bottleneck link rate in Mbps", cfg.bottleneckMbps);
    cmd.AddValue("accessMbps", "Access link rate in Mbps", cfg.accessMbps);
    cmd.AddValue("bgRateMbps", "Background offered rate in Mbps", cfg.bgRateMbps);
    cmd.AddValue("simTime", "Simulation time in seconds", cfg.simTime);
    cmd.AddValue("burstyBg", "Use bursty background traffic", cfg.burstyBg);
    cmd.Parse(argc, argv);

    const std::string outFile = "../results/summary.csv";

    // Traffic classes used in the study.
    // Rates/sizes are fixed here so the scheduler itself is doing most of the talking.
    std::vector<ClassSpec> classes = {
        {"control", 9000, 64, 200.0, 36, true}, // AF42 -> high priority band
        {"status", 10000, 96, 100.0, 38, false}, // AF43 -> middle priority band
        {"telemetry", 11000, 512, 50.0, 34, false}, // AF41 -> lower priority band
        {"bulk", 12000, 1400, 20.0, 8, false} // CS1 -> lowest/filler band
    };

    NodeContainer orchestrator;
    orchestrator.Create(1);

    NodeContainer routerA;
    routerA.Create(1);

    NodeContainer routerB;
    routerB.Create(1);

    NodeContainer followers;
    followers.Create(cfg.followers);

    NodeContainer bgSrc;
    bgSrc.Create(1);

    NodeContainer bgDst;
    bgDst.Create(1);

    NodeContainer bgSrcRev;
    bgSrcRev.Create(1);

    NodeContainer bgDstRev;
    bgDstRev.Create(1);

    InternetStackHelper internet;
    internet.Install(orchestrator);
    internet.Install(routerA);
    internet.Install(routerB);
    internet.Install(followers);
    internet.Install(bgSrc);
    internet.Install(bgDst);
    internet.Install(bgSrcRev);
    internet.Install(bgDstRev);
}
```

```
// Access links are intentionally generous.
// The real fight should happen at the shared bottleneck, not out on the edges.
PointToPointHelper access;
access.SetDeviceAttribute("DataRate", StringValue(std::to_string((uint32_t)cfg.accessMbps) + "Mbps"));
access.SetChannelAttribute("Delay", StringValue("1ms"));
access.SetQueue("ns3::DropTailQueue<Packet>", "MaxSize", StringValue("1p"));

PointToPointHelper bottleneck;
bottleneck.SetDeviceAttribute("DataRate", StringValue(std::to_string(cfg.bottleneckMbps) + "Mbps"));
bottleneck.SetChannelAttribute("Delay", StringValue("5ms"));
bottleneck.SetQueue("ns3::DropTailQueue<Packet>", "MaxSize", StringValue("1p"));

NetDeviceContainer devOrchA = access.Install(orchestrator.Get(0), routerA.Get(0));
NetDeviceContainer devBgSrcA = access.Install(bgSrc.Get(0), routerA.Get(0));
NetDeviceContainer devAB = bottleneck.Install(routerA.Get(0), routerB.Get(0));

std::vector<NetDeviceContainer> followerLinks;
followerLinks.reserve(cfg.followers);
for (uint32_t i = 0; i < cfg.followers; ++i)
{
    followerLinks.push_back(access.Install(routerB.Get(0), followers.Get(i)));
}
NetDeviceContainer devBgDstB = access.Install(routerB.Get(0), bgDst.Get(0));

NetDeviceContainer devBgSrcB = access.Install(routerB.Get(0), bgSrcRev.Get(0));
NetDeviceContainer devBgDstA = access.Install(routerA.Get(0), bgDstRev.Get(0));

Ipv4AddressHelper ipv4;

ipv4.SetBase("10.0.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifOrchA = ipv4.Assign(devOrchA);

ipv4.SetBase("10.1.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifBgSrcA = ipv4.Assign(devBgSrcA);

ipv4.SetBase("10.2.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifAB = ipv4.Assign(devAB);

std::vector<Ipv4InterfaceContainer> ifFollowerLinks;
for (uint32_t i = 0; i < cfg.followers; ++i)
{
    std::ostringstream subnet;
    subnet << "10.3." << i << ".0";
    ipv4.SetBase(subnet.str().c_str(), "255.255.255.0");
    ifFollowerLinks.push_back(ipv4.Assign(followerLinks[i]));
}

ipv4.SetBase("10.4.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifBgDstB = ipv4.Assign(devBgDstB);

ipv4.SetBase("10.5.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifBgSrcB = ipv4.Assign(devBgSrcB);

ipv4.SetBase("10.6.0.0", "255.255.255.0");
Ipv4InterfaceContainer ifBgDstA = ipv4.Assign(devBgDstA);

Ipv4GlobalRoutingHelper::PopulateRoutingTables();

// Install the root qdisc on both bottleneck-facing devices.
// I did not want the reverse path to be "free" by accident.
InstallRootQdiscOnDevice(devAB.Get(0), cfg.scheduler, cfg.queuePkts);
```

```
InstallRootQdiscOnDevice(devAB.Get(1), cfg.scheduler, cfg.queuePkts);

const uint16_t bgPort = 13000;
const uint16_t bgPortRev = 13001;

ApplicationContainer sinks;
ApplicationContainer apps;

// Spin up one set of apps per class, per follower.
// A little repetitive, but it keeps the traffic generation obvious.
for (const auto& cls : classes)
{
    for (uint32_t i = 0; i < cfg.followers; ++i)
    {
        uint16_t port = cls.portBase + i;

        if (cls.downlink)
        {
            PacketSinkHelper sink("ns3::UdpSocketFactory",
                                   InetSocketAddress(Ipv4Address::GetAny(), port));
            sinks.Add(sink.Install(followers.Get(i)));

            OnOffHelper src("ns3::UdpSocketFactory",
                            InetSocketAddress(ifFollowerLinks[i].GetAddress(1), port));
            src.SetAttribute("PacketSize", UintegerValue(cls.packetBytes));
            src.SetAttribute("DataRate",
                             DataRateValue(DataRate((uint64_t)(cls.packetBytes * 8.0 * cls.ratePps))));
            src.SetAttribute("Tos", UintegerValue(MakeTos(cfg.dscp, cls.dscpDecimal)));
            src.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=1]"));
            src.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));
            apps.Add(src.Install(orchestrator.Get(0)));
        }
        else
        {
            PacketSinkHelper sink("ns3::UdpSocketFactory",
                                   InetSocketAddress(Ipv4Address::GetAny(), port));
            sinks.Add(sink.Install(orchestrator.Get(0)));

            OnOffHelper src("ns3::UdpSocketFactory",
                            InetSocketAddress(ifOrchA.GetAddress(0), port));
            src.SetAttribute("PacketSize", UintegerValue(cls.packetBytes));
            src.SetAttribute("DataRate",
                             DataRateValue(DataRate((uint64_t)(cls.packetBytes * 8.0 * cls.ratePps))));
            src.SetAttribute("Tos", UintegerValue(MakeTos(cfg.dscp, cls.dscpDecimal)));
            src.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=1]"));
            src.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));
            apps.Add(src.Install(followers.Get(i)));
        }
    }
}

{
    // Forward background traffic to make sure the bottleneck is not just lightly loaded.
    PacketSinkHelper sink("ns3::UdpSocketFactory",
                           InetSocketAddress(Ipv4Address::GetAny(), bgPort));
    sinks.Add(sink.Install(bgDst.Get(0)));

    OnOffHelper bg("ns3::UdpSocketFactory",
                   InetSocketAddress(ifBgDstB.GetAddress(1), bgPort));
    bg.SetAttribute("PacketSize", UintegerValue(1400));
    bg.SetAttribute("DataRate", DataRateValue(DataRate((uint64_t)(cfg.bgRateMbps * 1000000.0))));
    bg.SetAttribute("Tos", UintegerValue(MakeTos(cfg.dscp, 0)));
}
```

```
if (cfg.burstyBg)
{
    bg.SetAttribute("OnTime", StringValue("ns3::ExponentialRandomVariable[Mean=0.15]"));
    bg.SetAttribute("OffTime", StringValue("ns3::ExponentialRandomVariable[Mean=0.05]"));
}
else
{
    bg.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=1]"));
    bg.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));
}

apps.Add(bg.Install(bgSrc.Get(0)));
}

{
    // Reverse background traffic too, since this study is supposed to stay bidirectional.
    PacketSinkHelper sink("ns3::UdpSocketFactory",
        InetSocketAddress(Ipv4Address::GetAny(), bgPortRev));
    sinks.Add(sink.Install(bgDstRev.Get(0)));

    OnOffHelper bgRev("ns3::UdpSocketFactory",
        InetSocketAddress(ifBgDstA.GetAddress(1), bgPortRev));
    bgRev.SetAttribute("PacketSize", UintegerValue(1400));
    bgRev.SetAttribute("DataRate", DataRateValue(DataRate((uint64_t)(cfg.bgRateMbps * 1000000.0))));
    bgRev.SetAttribute("Tos", UintegerValue(MakeTos(cfg.dscp, 0)));

    if (cfg.burstyBg)
    {
        bgRev.SetAttribute("OnTime", StringValue("ns3::ExponentialRandomVariable[Mean=0.15]"));
        bgRev.SetAttribute("OffTime", StringValue("ns3::ExponentialRandomVariable[Mean=0.05]"));
    }
    else
    {
        bgRev.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=1]"));
        bgRev.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));
    }

    apps.Add(bgRev.Install(bgSrcRev.Get(0)));
}

sinks.Start(Seconds(0.0));
sinks.Stop(Seconds(cfg.simTime + 1.0));

apps.Start(Seconds(1.0));
apps.Stop(Seconds(cfg.simTime));

FlowMonitorHelper flowHelper;
Ptr<FlowMonitor> flowMon = flowHelper.InstallAll();

Simulator::Stop(Seconds(cfg.simTime + 1.0));
Simulator::Run();

flowMon->CheckForLostPackets();
Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowHelper.GetClassifier());
auto stats = flowMon->GetFlowStats();

bool writeHeader = false;
{
    std::ifstream testRead(outFile);
    writeHeader = !testRead.good() || testRead.peek() == std::ifstream::traits_type::eof();
}

std::ofstream out(outFile, std::ios::app);
```

```
if (!out.is_open())
{
    std::cerr << "Failed to open CSV output: " << outFile << std::endl;
    Simulator::Destroy();
    return 1;
}

if (writeHeader)
{
    out << "run_id,seed,scenario,scheduler_key,scheduler_label,dscp_mode,followers,bottleneck_mbps,queue_pkts,"
        "bg_rate_mbps,bursty_bg,flow_class,src,dst,src_port,dst_port,tx_packets,rx_packets,loss_pct,"
        "throughput_bps,delay_mean_ms,jitter_mean_ms\n";
}

// Apps start at 1 s, so this is the window I use for throughput math.
// It is a simple choice, but at least it is consistent across all runs.
const double activeDuration = cfg.simTime - 1.0;

for (const auto& [flowId, st] : stats)
{
    auto t = classifier->FindFlow(flowId);

    // Figure out which logical class this flow belongs to from its destination port.
    // A bit old-school, but good enough for this experiment.
    std::string flowClass = "other";

    for (const auto& cls : classes)
    {
        if (t.destinationPort >= cls.portBase && t.destinationPort < cls.portBase + cfg.followers)
        {
            flowClass = cls.name;
            break;
        }
    }

    if (t.destinationPort == bgPort || t.destinationPort == bgPortRev)
    {
        flowClass = "background";
    }

    if (flowClass == "other")
    {
        continue;
    }

    double lossPct = 0.0;
    if (st.txPackets > 0)
    {
        lossPct = 100.0 * (double)(st.txPackets - st.rxPackets) / (double)st.txPackets;
    }

    double throughputBps = 0.0;
    if (activeDuration > 0)
    {
        throughputBps = (st.rxBytes * 8.0) / activeDuration;
    }

    double delayMeanMs = 0.0;
    if (st.rxPackets > 0)
    {
        delayMeanMs = 1000.0 * st.delaySum.GetSeconds() / st.rxPackets;
    }
}
```

```
double jitterMeanMs = 0.0;
if (st.rxPackets > 1)
{
    jitterMeanMs = 1000.0 * st.jitterSum.GetSeconds() / (st.rxPackets - 1);
}

out << cfg.runId << ","
    << RngSeedManager::GetRun() << ","
    << cfg.scenario << ","
    << cfg.scheduler << ","
    << "\"" << SchedulerLabel(cfg.scheduler) << "\"" << ","
    << (cfg.dscp ? "on" : "off") << ","
    << cfg.followers << ","
    << std::fixed << std::setprecision(3) << cfg.bottleneckMbps << ","
    << cfg.queuePkts << ","
    << cfg.bgRateMbps << ","
    << (cfg.burstyBg ? "yes" : "no") << ","
    << flowClass << ","
    << t.sourceAddress << ","
    << t.destinationAddress << ","
    << t.sourcePort << ","
    << t.destinationPort << ","
    << st.txPackets << ","
    << st.rxPackets << ","
    << std::fixed << std::setprecision(4) << lossPct << ","
    << throughputBps << ","
    << delayMeanMs << ","
    << jitterMeanMs << "\n";
}

out.close();
Simulator::Destroy();
return 0;
}
```