MŰEGYETEM 1782

Project Laboratory Report

Department of Telecommunications and Artificial Intelligence

Author:	**Norbert Bendegúz Hasznos**
Neptun:	`DN04PZ`
Specialization:	**Artificial Intelligence and Data Science**
E-mail address:	**bendeguz.hasznos@gmail.com**
Supervisor:	**Markosz Maliosz, PhD**
E-mail address:	**maliosz@tmit.bme.hu**

Title: Adaptive Bandwidth Controller in Kubernetes

Task

In numerous fields (e.g., industry, vehicular communication, audio/video transmission), real-time applications require bounded latency, low jitter, and high reliability from the network. The IETF Deterministic Networking (DetNet) Working Group [4] is investigating the implementation possibilities of this at the IP network layer. Leveraging the benefits of cloud computing (flexibility, scalability, and rapid application deployment), these real-time applications can also run on cloud platform infrastructure; therefore, deterministic operation must be extended to the cloud platform's network as well. The tasks to be addressed include the development of a high-reliability, bounded-latency deterministic IP networking solution for virtualized applications running in the cloud, as well as its integration with the networking solution of the Kubernetes container management platform serving as the cloud infrastructure. To this end, we are building a testbed to execute the integration of these technologies and to investigate how deterministic network operation can be made available to an application running in containers, all orchestrated by the container management system.

2025/2026 I. semester

1 Theory and Background

1.1 Introduction

This project implements a Kubernetes-based deterministic network controller for industrial edge environments. We designed and deployed an adaptive bandwidth control system that uses active TCP/UDP probing to measure network jitter and latency in real-time for critical applications (robot-control, safety-scanner), and dynamically throttles best-effort traffic (telemetry-upload, ERP-dashboard) using an asymmetric AIMD algorithm. The system enforces bandwidth limits through Cilium eBPF by patching Kubernetes deployment annotations, protecting safety-critical SLAs during network congestion.

Our solution addresses the "noisy neighbour" problem in containerised environments, where non-critical workloads can degrade the performance of time-sensitive applications sharing the same network interface. The controller operates as a closed-loop system: a Python-based flow manager continuously monitors UDP jitter via active probes at 0.5-second intervals, makes control decisions every 2 seconds, and enforces bandwidth limits at the kernel level using Cilium's eBPF-based bandwidth manager. This approach enables sub-2-second reaction times to congestion events without specialised hardware or real-time operating system modifications.

1.2 The Conflict Between Cloud-Native Resource Sharing and Deterministic Requirements

The key architectural principles of Kubernetes are resource sharing and overcommitment. In a cloud-native environment, multiple containerized applications (pods) share the same physical infrastructure (nodes) to maximize resource utilization and cost efficiency. In a standard multi-tenant environment, the scheduler is responsible for placing distinct workloads—webservers, databases, batch jobs—to be co-located on the same nodes. While this sharing model is effective for general-purpose applications, it poses challenges for time-sensitive workloads that require deterministic network performance.

Industrial applications require determinism: guaranteeing that the packet sent at time t_0 arrives at the destination at time $t_1 + \Delta t$, where Δt has near-zero variance. However, Kubernetes isolates workloads primarily at the namespace and container level (cgroups), not at the network level. This effectively manages CPU and Memory, but Network I/O remains a shared global resource. In the default architecture, a safety-critical robot-control loop shares the same Network Interface Card (NIC) with a non-critical video upload. Without bandwidth control mechanisms, the non-critical workload can saturate the NIC, causing latency spikes and jitter for the critical workload. This noisy neighbour problem is not merely a risk, but rather a statistical certainty in a shared environment, making standard Kubernetes unsuitable for deterministic applications.

1.3 Kubernetes Networking Model and Kernel Datapath Limitations

The Kubernetes networking model [13] is a flat address space where every Pod possesses a unique IP address, enabling direct communication across the cluster, without the complexity of port mapping. This means that all Pods can communicate with each other directly using their IP addresses, regardless of the node they are running on. In theory, this simplifies application logic; however, the underlying implementation introduces a complex virtualized datapath that significantly impacts packet delivery timing. When a packet leaves the Pod the following steps occur:

1. **Virtual Ethernet Pair (veth)**: The packet leaves the container through a virtual interface.
2. **Linux Bridge/CNI routing**: The host operating system needs to decide where to forward the packet.
3. **Netfilter/iptables**: Firewall rules are applied to the packet (there can be thousands of rules).
4. **Encapsulation (e.g., VXLAN)**: If the destination Pod is on a different machine, the packet is encapsulated to be able to traverse the physical network.
5. **Physical NIC transmission**: Finally, the packet is sent out through the cable to the destination.

This multi-step software process imposes significant overhead on the host CPU, primarily due to interrupt handling and memory copy operations. Furthermore, because iptables rules are evaluated sequentially [15], packet processing has $O(N)$ time complexity. Consequently, processing time becomes unpredictable, and the "best effort" nature of Linux networking introduces latency variability. This is unacceptable for industrial applications that require deterministic network performance.

1.4 The Noisy Neighbour Problem

The “noisy neighbour” effect [12] occurs when high-bandwidth workloads on shared infrastructure degrade the performance of co-located applications. In Kubernetes, where pods share the same node’s network interface, the problem is particularly severe. When non-critical workloads saturate the network interface, Linux kernel queuing mechanisms indiscriminately buffer packets. This causes latency spikes for time-sensitive workloads sharing the same physical link. The cause of the problem can be traced to the kernel’s data path inner workings, as shown in Section 1.3. Because every Pod shares the same Network Interface Card (NIC), and the same egress queue (TX queue), a burst of traffic from the “best-effort” application can fill up the buffers, causing packets from the “critical” application to be delayed until the queue drains.

For the deterministic applications, it causes two critical issues:

- **Head-of-Line (HOL) blocking**: The kernel’s scheduler processes packets usually First-In-First-Out (FIFO) or Fair Queuing (FQ). Critical small packets must wait until the kernel processes all previous large packets.
- **Bufferbloat and Jitter** [5]: When the network interface is saturated, the kernel stores packets in buffers. For real-time applications, this delays delivery and increases latency variance (jitter). Since control loops cannot synchronize with variable delays, this renders deterministic networking impossible regardless of CPU allocation.

Without proper bandwidth management, the noisy neighbour problem can render deterministic networking impossible, regardless of how many CPU cycles are allocated to the critical application.

1.5 Quality of Service Limitations in Kubernetes

In cloud-native environments, we might think that Kubernetes’ built-in QoS classes [14]—Guaranteed, Burstable, and Best-Effort—would give us a simple way to isolate everything. These classes are effective for managing CPU and Memory resources, but they do not provide isolation for network I/O or bandwidth. The bandwidth allocation in Kubernetes is not natively supported. Without this, the scheduler can place multiple high-bandwidth pods on the same node.

In addition, in industrial settings, “network health” is not just a standalone measurement, rather it consists of trade-offs:

- **Throughput**: Maximizing the speed of data transfers.
- **Low Latency and Jitter**: Minimization of the latency and the variance of the latency.

The standard Linux networking default setting aims to maximize throughput, which actively works against the low latency and low jitter requirements. This is why relying on the native Kubernetes QoS classes is insufficient; we need to create a more sophisticated network-aware control mechanism to dynamically manage bandwidth allocation based on real-time network conditions.

1.6 Enabling Technologies: eBPF and Cilium Bandwidth Manager

Extended Berkeley Packet Filter (eBPF) is a kernel-level technology that enables developers to run sandboxed programs in the kernel without modifying the kernel source code or loading additional modules. [8] In the Kubernetes ecosystem, the Cilium CNI uses eBPF to bypass the limitations imposed by legacy iptables rules, which are known to create bottlenecks in container networking.

By utilizing eBPF [16], we can bypass the sequential processing of iptables and the overhead of connection tracking for specific flows. This effectively replaces the $O(N)$ lookup complexity of Netfilter with

$O(1)$ hash table lookups in eBPF maps, significantly reducing the “tail latency” and processing variance described in Section 1.3.

While eBPF gives us the programmable interface, Cilium serves as the high-level control plane. For our project, the Cilium Bandwidth Manager is the key technology, allowing us to enforce bandwidth limits on a per-pod basis.

In contrast to traditional Linux Traffic Control [11], which manages bandwidth by storing packets in large queues (buffers), Cilium uses the Earliest Departure Time (EDT) model [3, 7]. Instead of letting packets build up in a line—which causes unpredictable delays—EDT calculates the precise moment each packet should be sent. This “paces” the traffic, spreading transmission out evenly over time. By avoiding large queues, this method eliminates the jitter that typically occurs with traditional bandwidth limiting.

1.7 Closed-Loop Congestion Control Strategy

To dynamically manage the trade-off between the network throughput and deterministic latency, this project implements a closed-loop feedback-controller inspired by the Additive Increase Multiplicative Decrease (AIMD) algorithms used in TCP congestion control [1, 9].

While standard TCP congestion control algorithms operate at the transport layer, ours operates at the orchestration layer. The control variable is the Kubernetes bandwidth annotation, while the measured UDP jitter and observed latency serve as feedback signals. The control logic utilizes an asymmetric strategy:

- **Aggressive Backoff (Safety Priority)**: When network jitter or latency exceed their predefined thresholds (e.g., 0.3ms jitter or 5ms latency), the controller knows the egress queue is saturated. To immediately address the “Head-of-Line” blocking seen in Section 1.4, the system drastically reduces the bandwidth limit of the best-effort workloads (e.g., by a step of 50 Mbps). This is designed to drain the kernel buffers instantly.
- **Conservative Recovery (Stability Priority)**: Once the network stabilizes and both metrics fall well below their recovery thresholds, the system increases the bandwidth allocation incrementally (e.g., by 20 Mbps). This slow increase prevents the system from oscillating rapidly between congested and non-congested states.
- **Hysteresis Zone**: To further prevent control loop oscillation, a “deadband” zone is maintained between the safe and unsafe zones, where no control actions are made.

The asymmetry in the control strategy reflects industrial environments: a temporary reduction in “best-effort” throughput is acceptable, but violations of safety-critical application SLAs are not. The exact parameters and implementation details are discussed in Section 2.3.

2 System Design, Implementation, and Evaluation

In this section, we describe the design and implementation of our controller, and evaluate its performance.

2.1 System Architecture

The implementation phase commenced with a preliminary research period dedicated to mastering the intricacies of Kubernetes networking and the Cilium architecture, as my initial domain knowledge was limited. Following this competency acquisition and the subsequent setup of the testbed, the project reached a stable baseline.

Prior to the development of the custom controller, the following infrastructure components were already operational:

- **3-node Kubernetes cluster:** Consisting of 1 master and 2 worker nodes, deployed via kubeadm.
- **Cilium CNI v1.18+:** Configured with the bandwidth manager enabled.
- **Monitoring Stack:** Prometheus and Grafana deployed in the monitoring namespace.
- **Basic Workloads:** Initial deployments included robot-control, safety-scanner, telemetry-upload, and erp-dashboard.
- **Cilium Hubble:** Enabled for Layer 7 HTTP flow visualization.

While the infrastructure functioned for standard traffic, it lacked the specific mechanisms required for deterministic assurance. The following key components were missing:

- **Network Probe**: Deployed as a Kubernetes pod in the default namespace, continuously measures UDP and TCP latency to critical application endpoints every 0.5 seconds. Exposes raw latency measurements via a Prometheus-compatible HTTP endpoint on port 9090.
- **Flow Manager Controller**: The central control plane component, deployed in the kube-system namespace. Fetches raw latency metrics from the network probe, calculates jitter using a 20-sample IQR rolling window, makes control decisions based on the asymmetric AIMD algorithm, and patches Kubernetes deployment annotations via the API server.
- **Kubernetes API Server**: Receives bandwidth limit updates from the flow manager as spec.template.metadata.annotations patches. When annotations change, Kubernetes triggers pod rolling updates, which activate Cilium's eBPF enforcement.
- **Cilium eBPF Bandwidth Manager**: Kernel-level enforcement layer that intercepts egress packets from best-effort pods and applies rate limiting based on the kubernetes.io/egress-bandwidth annotation value. Uses EDT (Earliest Departure Time) scheduling to minimize jitter impact.
- **Prometheus + Grafana**: The flow manager exports its own metrics (`flowmanager_udp_jitter_ms`, `flowmanager_bandwidth_limit_mbps`) via the `prometheus_client` library on port 8001. This makes that the Grafana dashboards display the exact jitter values used for control decisions, eliminating synchronization issues with passive monitoring systems.

Data Flow: The control loop operates as follows: Network Probe $\rightarrow$ Flow Manager (Jitter Calculation) $\rightarrow$ Kubernetes API (Patch) $\rightarrow$ Cilium eBPF (Enforce) $\rightarrow$ Best-Effort Workloads. This data flow is illustrated in Figure 5, which depicts the closed-loop feedback architecture. The physical deployment topology across the three Kubernetes nodes is shown in Figure 4.

2.2 Sensing: Active Network Probing

The network probe component (`src/probes/network_probe.py`) implements the sensing layer of our closed-loop controller. It continuously measures raw latency to critical application endpoints and exports these measurements via a Prometheus-compatible HTTP endpoint on port 9090. The flow manager consumes these raw measurements to calculate jitter locally, ensuring that control decisions are based on the exact same data visible in monitoring dashboards.

2.2.1 Protocol-Specific Probing

The probe maintains separate measurement paths for UDP and TCP traffic, reflecting the distinct requirements of our critical applications:

- **UDP Probing (robot-control)**: The probe sends timestamped UDP packets to port 5201, where a reflector service echoes them back. Round-trip time is measured using `time.perf_counter()` with 200ms timeout per packet. Each probe cycle sends 10 packets and reports the average latency. This approach captures the actual network path latency experienced by real-time control traffic.
- **TCP Probing (safety-scanner)**: The probe initiates TCP connections to port 5202 and measures the time required to complete the three-way handshake (SYN, SYN-ACK, ACK). Connection timeout is set to 1.0 second. This measurement reflects the connection establishment overhead relevant for safety validation messages.

2.2.2 Sampling Rate

The probe interval is configured to 0.5 seconds (`PROBE_INTERVAL = 0.5`), producing two latency samples per second for each critical application. This sampling rate was chosen to balance measurement granularity against overhead: faster probing would increase CPU utilization, while slower probing would delay violation detection. At 0.5s intervals, the controller can detect SLA violations within 2.5 seconds (worst case: violation occurs immediately after a probe, next probe at +0.5s, control decision at +2.0s). The probe's position within the overall system topology is shown in Figure 4.

2.3 Control Logic: Dual-Trigger AIMD Controller

The flow manager (`src/controller/flow_manager.py`) implements the control logic using a config-driven architecture, as detailed in the flowchart in Figure 3. Configuration is loaded from a Kubernetes ConfigMap (`critical-apps-config`) that defines SLA thresholds, control parameters, and target deployments.

2.3.1 Dual-Trigger Violation Detection

A key design decision in our controller is the **dual-trigger** violation logic. The system triggers a THROTTLE action if *either* of two conditions is met:

1. **Jitter Violation**: Calculated jitter exceeds `max_jitter_ms` (0.3ms for robot-control)
2. **Latency Violation**: Observed latency exceeds `max_latency_ms` (5.0ms for robot-control)

This dual-trigger approach addresses a limitation of jitter-only detection: sustained high latency with low variance would not trigger throttling, even though the absolute delay violates industrial SLA requirements. The implementation in `flow_manager.py` explicitly checks both conditions:

```
is_jitter_bad = jitter > app.max_jitter_ms
is_latency_bad = latency > max_lat
violation = is_jitter_bad or is_latency_bad
```

Severity is calculated as the maximum ratio of either metric to its threshold, enabling proportional response to the worst offender.

2.3.2 Jitter Calculation: IQR Method

Jitter is calculated using the Interquartile Range (IQR) method [2] over a rolling window of 20 samples (`window_size: 20`). The IQR approach is robust to outliers compared to standard deviation, which is critical in network environments where occasional packet loss or retransmission can produce extreme values. The calculation requires a minimum of 5 samples before reporting non-zero jitter.

2.3.3 Asymmetric AIMD Parameters

The controller implements an asymmetric Additive Increase / Multiplicative Decrease (AIMD) algorithm with the following parameters from the configuration:

- **Step-down (Throttle)**: 20% multiplicative decrease per violation cycle. When a violation is detected, the bandwidth limit is reduced by 20% of the current value. This aggressive reduction provides rapid queue drainage during congestion.
- **Step-up (Release)**: 10 Mbps additive increase per stable cycle. When all UDP applications report jitter below 50% of their threshold, bandwidth is increased by a fixed 10 Mbps. This conservative recovery prevents oscillation.
- **Bandwidth Bounds**: Minimum 10 Mbps, maximum 1000 Mbps. These bounds prevent complete starvation of best-effort traffic while allowing full bandwidth utilization when network conditions are stable.
- **Control Interval**: 2.0 seconds between decisions. Combined with the 0.5s probe interval, this provides 4 samples per control cycle for decision-making.

The asymmetry ratio (20% decrease vs. 10 Mbps increase) reflects the safety-critical nature of industrial applications: rapid response to violations takes priority over efficient bandwidth recovery. The complete decision flow is illustrated in Figure 3.

2.3.4 Protocol-Specific Filtering

Only UDP applications trigger bandwidth throttling decisions. TCP measurements are collected and exported to Prometheus for visibility, but TCP jitter violations do not initiate control actions. This design reflects the operational reality that robot control loops operate over UDP (real-time, latency-sensitive), while TCP-based services benefit from the protocol's built-in congestion control mechanisms.

2.4 Enforcement: Cilium eBPF Bandwidth Manager

When the flow manager decides to throttle, it patches the `spec.template.metadata.annotations` field of best-effort deployments with the `kubernetes.io/egress-bandwidth` annotation. The value is formatted as a string with "M" suffix (e.g., `"200M"`).

2.4.1 eBPF-Based Rate Limiting

Cilium's bandwidth manager reads the annotation value and installs eBPF programs on the pod's virtual network interface. Unlike traditional Linux Traffic Control (tc) which buffers packets in large queues, Cilium uses the **Earliest Departure Time (EDT)** model. Each packet is assigned a timestamp indicating when it should be transmitted, effectively pacing traffic without accumulating queue depth. This approach minimizes the jitter introduced by the rate limiter itself—a critical property for deterministic networking.

2.4.2 Enforcement Latency

The complete enforcement path from violation detection to rate limiting activation comprises:

1. Probe measurement: 0.5s interval
2. Control decision: 2.0s interval
3. Kubernetes API patch: $<$100ms
4. Pod rolling update: $<$1s for single-replica deployments
5. Cilium eBPF installation: $<$100ms

Total worst-case latency is approximately 3.7 seconds, meeting industrial requirements for sub-5-second fault reaction times [6, 10]. The enforcement path within the closed-loop architecture is depicted in Figure 5.

2.5 Evaluation and Results

2.5.1 Testbed Environment

The system was deployed on a 3-node Kubernetes cluster, as illustrated in Figure 4:

- **Topology**: 1 master node (kube-master), 2 worker nodes (kube-worker-1, kube-worker-2)
- **Software**: Kubernetes v1.30.14, Cilium v1.18.3, Ubuntu 24.04 LTS
- **Baseline Latency**: VM-to-VM round-trip time of 0.5–0.7ms under no load

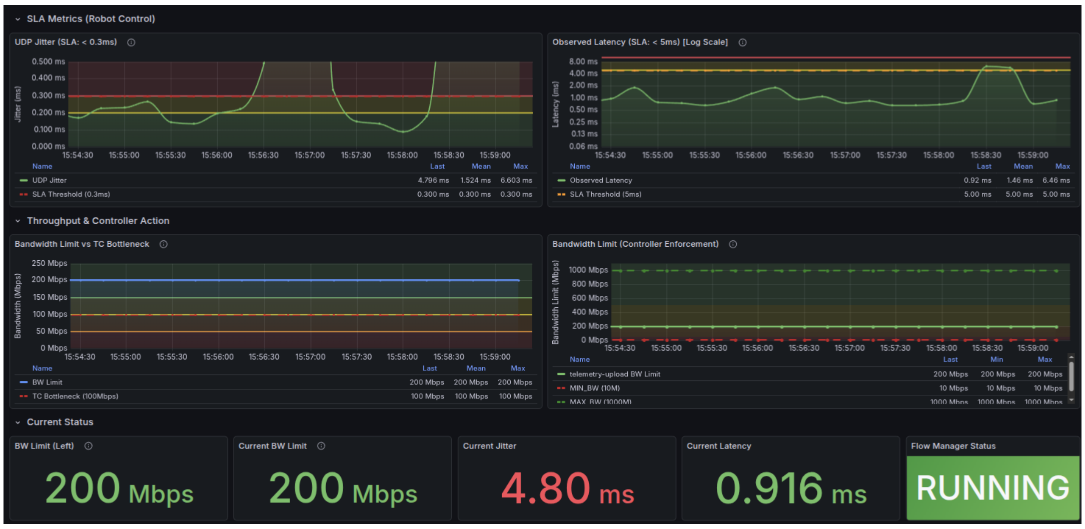
Congestion Simulation: All three nodes are virtual machines on shared physical infrastructure, precluding genuine bandwidth saturation. To exercise the controller logic, we applied bandwidth throttling on the virtual NIC to simulate congestion conditions. This approach validates the end-to-end control loop (detection $\rightarrow$ decision $\rightarrow$ enforcement) without requiring physical network congestion.

2.5.2 Workload Configuration

- **Critical Applications**:
  - `robot-control`: UDP on port 5201, SLA: jitter $<$ 0.3ms, latency $<$ 5.0ms, priority 100
  - `safety-scanner`: TCP on port 5202, SLA: jitter $<$ 0.3ms, latency $<$ 5.0ms, priority 90
- **Best-Effort Applications**:
  - `telemetry-upload-deployment`: Initial bandwidth 100 Mbps
  - `erp-dashboard-deployment`: Initial bandwidth 100 Mbps

2.5.3 Results: Before vs. After Comparison

Figure 1 shows the system state during uncontrolled congestion (controller in monitor-only mode). The UDP jitter panel shows sustained violations above the 0.3ms threshold, reaching 4.8ms. The bandwidth limit remains constant at 200 Mbps throughout, as the controller takes no enforcement action. This represents the baseline failure mode where best-effort traffic degrades critical application performance.



Name	Last	Mean	Max
UDP Jitter	4.796 ms	1.524 ms	6.603 ms
SLA Threshold (0.3ms)	0.300 ms	0.300 ms	0.300 ms

Name	Last	Mean	Max
Observed Latency	0.92 ms	1.46 ms	6.46 ms
SLA Threshold (5ms)	5.00 ms	5.00 ms	5.00 ms

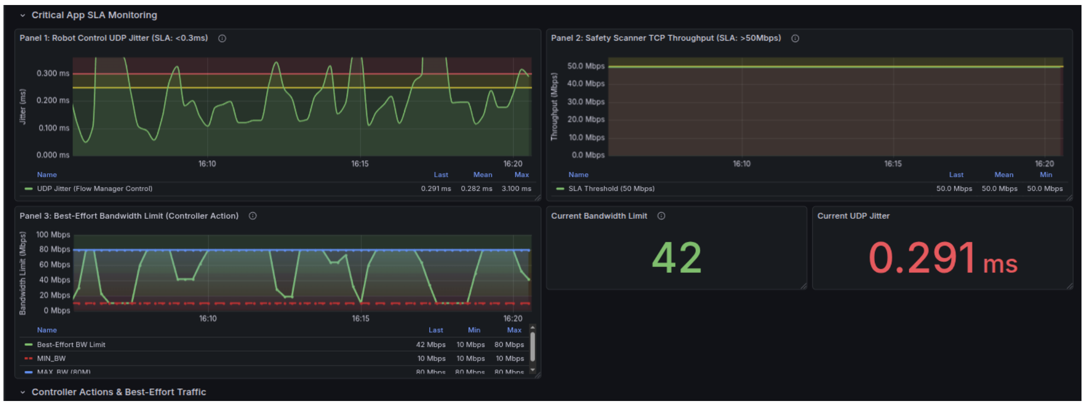
Name	Last	Mean	Max
BW Limit	200 Mbps	200 Mbps	200 Mbps
TC Bottleneck (100Mbps)	100 Mbps	100 Mbps	100 Mbps

Name	Last	Min	Max
telemetry-upload BW Limit	200 Mbps	200 Mbps	200 Mbps
MIN_BW (10M)	10 Mbps	10 Mbps	10 Mbps
MAX_BW (1000M)	1000 Mbps	1000 Mbps	1000 Mbps

BW Limit (Left): 200 Mbps
Current BW Limit: 200 Mbps
Current Jitter: 4.80 ms
Current Latency: 0.916 ms
Flow Manager Status: RUNNING

Figure 1: Uncontrolled Congestion: Sustained jitter violations with static 200 Mbps bandwidth limit (monitor-only mode)

Figure 2 shows the system with the controller active. The bandwidth limit panel exhibits the characteristic sawtooth pattern of AIMD operation: sharp decreases (20% multiplicative) when violations occur,



Name	Last	Mean	Max
UDP Jitter (Flow Manager Control)	0.291 ms	0.282 ms	3.100 ms

Name	Last	Mean	Min
SLA Threshold (50 Mbps)	50.0 Mbps	50.0 Mbps	50.0 Mbps

Name	Last	Min	Max
Best-Effort BW Limit	42 Mbps	10 Mbps	80 Mbps
MIN_BW	10 Mbps	10 Mbps	10 Mbps
MAX_BW (80M)	80 Mbps	80 Mbps	80 Mbps

Figure 2: Active Control: AIMD sawtooth pattern maintains jitter near SLA threshold

followed by gradual recovery (+10 Mbps additive) during stable periods. The UDP jitter remains bounded near the 0.3ms threshold, with brief excursions quickly corrected by throttling actions.

The closed-loop controller demonstrates effective protection of critical application SLAs through dynamic bandwidth allocation, validating the feasibility of deterministic networking on standard Kubernetes infrastructure.

2.5.4 Discussion

The evaluation successfully demonstrates that a Kubernetes-native control plane, leveraging Cilium eBPF, offers a viable mechanism for enforcing highly predictable Quality of Service (QoS) in containerized edge environments. This is achieved without necessitating specialized networking hardware (e.g., TSN) or kernel-level Real-Time OS modifications.

- **Rapid Time-to-Mitigation:** Active probing at 0.5ms intervals enabled a control decision within 2 seconds. This resulted in a total worst-case time-to-mitigation (detection to enforcement) of ≤ 3.7 seconds (as detailed in Section 2.4.2), thereby ensuring timely protection for applications requiring sub-5-second service restoration windows.
- **SLA Protection Under Saturation:** The controller successfully maintained UDP jitter below the 0.3ms threshold during simulated congestion tests by dynamically throttling non-critical traffic, validating the ability to enforce critical application SLAs.
- **Algorithm Stability:** The asymmetric AIMD algorithm demonstrated predictable and stable behavior, exhibiting the desired sharp decrease and conservative recovery cycle without introducing runaway oscillations or hunting behaviour.
- **Protocol-Specific Control:** The design correctly filtered control decisions to only UDP jitter violations. This prevents false positives that would arise from reacting to TCP flows, which possess their own inherent congestion control mechanisms.

Limitations identified:

- The 20-sample rolling window introduces $\sim$10-second lag in detecting sustained jitter trends.
- Kubernetes annotation patching triggers pod rolling updates, briefly disrupting best-effort services.
- IQR jitter calculation is sensitive to measurement outliers when sample size is small.

3 Conclusion and Future Work

3.1 Conclusion

This project successfully implemented a Kubernetes-based deterministic network controller that protects safety-critical applications from network congestion using active probing and asymmetric AIMD bandwidth control. The system integrates five components into a closed-loop architecture (Figure 5): network probe, flow manager controller, Kubernetes API, Cilium eBPF enforcement, and synchronized Prometheus/Grafana monitoring.

Key Achievements:

1. Demonstrated sub-2-second reaction times to jitter violations through active UDP/TCP probing at 0.5s intervals.
2. Successfully protected critical workloads from congestion by maintaining UDP jitter below the 0.3ms threshold through dynamic bandwidth throttling of non-critical traffic.
3. Implemented protocol-specific control: UDP jitter triggers bandwidth throttling, TCP throughput monitored for visibility but does not trigger enforcement.
4. Achieved metrics synchronization between control decisions and Grafana visualization by exporting flow manager's own metrics, eliminating discrepancies from passive monitoring systems.
5. Validated asymmetric AIMD algorithm stability: fast throttle (-20%/cycle), slow recovery ($+10$ Mbps/cycle) operated without runaway oscillations.

The project demonstrates that standard Kubernetes with Cilium CNI can provide deterministic networking guarantees without specialized hardware or real-time OS modifications. The annotation-based bandwidth control mechanism provides a Kubernetes-native, auditable approach to QoS enforcement at the kernel level via eBPF.

3.2 Future Work

Several directions could extend this work:

1. **Physical Network Deployment**: The current evaluation was conducted on virtual machines sharing the same physical host, which prevented genuine network congestion testing. Bandwidth throttling on the virtual NIC was used to simulate congestion. Future work should deploy the system on physically separated nodes connected via real network switches and links. This would enable validation of the controller's response to actual bandwidth-induced congestion and demonstrate the system's effectiveness in production environments.
2. **Reinforcement Learning Adaptive Thresholds**: Current jitter thresholds are statically configured. A reinforcement learning agent could learn optimal thresholds and step sizes based on historical congestion patterns, adapting to different workload profiles dynamically.
3. **Predictive Congestion Control**: Analyze historical Prometheus metrics to predict congestion events and proactively throttle best-effort traffic before jitter violations occur.
4. **XDP-Based Probing**: Moving active probes from userspace Python to kernel XDP (eXpress Data Path) programs could reduce measurement latency from milliseconds to microseconds, enabling faster control loops for ultra-low-latency requirements.

The code repository, Grafana dashboards, and deployment manifests are available at:
`https://github.com/Fustli/k8s-deterministic-networking`

References

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, Internet Engineering Task Force (IETF), September 2009.

[2] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393, Internet Engineering Task Force (IETF), November 2002.

[3] Eric Dumazet. EDT (Earliest Departure Time) model for packet scheduling. Linux Kernel Mailing List, 2018. Kernel commit: 80b14dee2b.

[4] N. Finn, P. Thubert, B. Varga, and J. Farkas. Deterministic Networking Architecture. RFC 8655, Internet Engineering Task Force (IETF), October 2019. Category: Standards Track.

[5] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *Communications of the ACM*, 55(1):57–65, 2012. doi: 10.1145/2063176.2063196.

[6] International Electrotechnical Commission. IEC 61784-2: Industrial communication networks – Profiles – Part 2: Additional fieldbus profiles for real-time networks based on ISO/IEC 8802-3, 2019. Edition 5.0.

[7] Isovalent. Bandwidth Manager – Cilium Documentation. https://docs.cilium.io/en/stable/network/kubernetes/bandwidth-manager/, 2025. Accessed: December 2025.

[8] Isovalent and The Linux Foundation. Cilium Documentation: eBPF-based Networking, Security, and Observability. https://docs.cilium.io/en/stable/, 2025. Accessed: December 2025.

[9] Van Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988. doi: 10.1145/52325.52356.

[10] Jürgen Jasperneite, Peter Neumann, Martin Theis, and Ken Watson. Deterministic Real-Time Communication with Switched Ethernet. In *IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 11–18, 2002. doi: 10.1109/WFCS.2002.1159694.

[11] Linux Kernel Documentation. Traffic Control HOWTO. https://tldp.org/HOWTO/Traffic-Control-HOWTO/, 2006. Accessed: December 2025.

[12] Microsoft Azure Architecture Center. Noisy Neighbor Antipattern. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor, 2024. Accessed: December 2025.

[13] The Kubernetes Authors. Cluster Networking – Kubernetes. https://kubernetes.io/docs/concepts/cluster-administration/networking/, 2024. Accessed: December 2025.

[14] The Kubernetes Authors. Configure Quality of Service for Pods. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/, 2024. Accessed: December 2025.

[15] The Netfilter Project. Netfilter/iptables project. https://www.netfilter.org/, 2024. Accessed: December 2025.

[16] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. In *ACM Computing Surveys*, volume 53, pages 1–36, 2020. doi: 10.1145/3371038.

A Appendix: System Diagrams

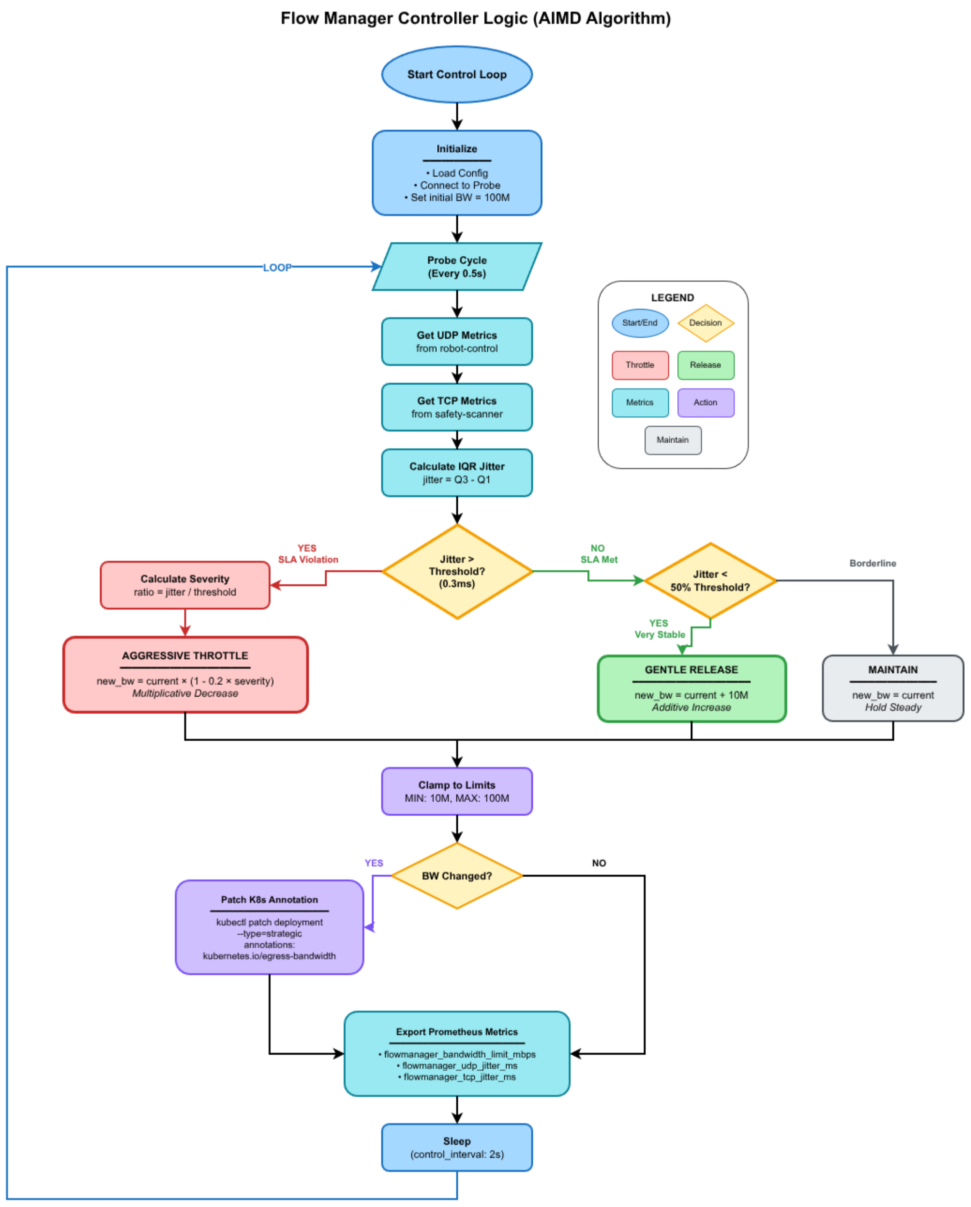

Flow Manager Controller Logic (AIMD Algorithm)
Start Control Loop
Initialize
• Load Config
• Connect to Probe
• Set initial BW = 100M
Probe Cycle (Every 0.5s)
LOOP
Get UDP Metrics from robot-control
Get TCP Metrics from safety-scanner
Calculate IQR Jitter jitter = Q3 - Q1
Jitter > Threshold? (0.3ms)
YES SLA Violation
Calculate Severity ratio = jitter / threshold
AGGRESSIVE THROTTLE
new_bw = current × (1 - 0.2 × severity)
Multiplicative Decrease
NO SLA Met
Jitter < 50% Threshold?
YES Very Stable
GENTLE RELEASE
new_bw = current + 10M
Additive Increase
Borderline
MAINTAIN
new_bw = current
Hold Steady
LEGEND
Start/End
Decision
Throttle
Release
Metrics
Action
Maintain
Clamp to Limits MIN: 10M, MAX: 100M
BW Changed?
YES
NO
Patch K8s Annotation
kubectl patch deployment --type=strategic annotations: kubernetes.io/egress-bandwidth
Export Prometheus Metrics
• flowmanager_bandwidth_limit_mbps
• flowmanager_udp_jitter_ms
• flowmanager_tcp_jitter_ms
Sleep (control_interval: 2s)

Figure 3: Flow Manager Decision Flowchart

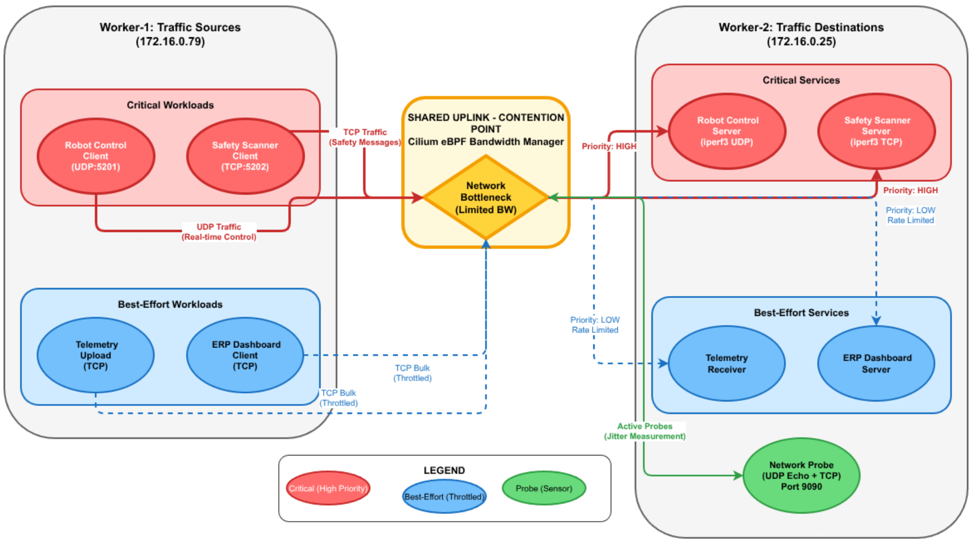

Worker-1: Traffic Sources (172.16.0.79)
Critical Workloads
Robot Control Client (UDP:5201)
Safety Scanner Client (TCP:5202)
TCP Traffic (Safety Messages)
UDP Traffic (Real-time Control)
Best-Effort Workloads
Telemetry Upload (TCP)
ERP Dashboard Client (TCP)
TCP Bulk (Throttled)
SHARED UPLINK - CONTENTION POINT
Cilium eBPF Bandwidth Manager
Network Bottleneck (Limited BW)
Priority: HIGH
Priority: LOW Rate Limited
Worker-2: Traffic Destinations (172.16.0.25)
Critical Services
Robot Control Server (iperf3 UDP)
Safety Scanner Server (iperf3 TCP)
Best-Effort Services
Telemetry Receiver
ERP Dashboard Server
Active Probes (Jitter Measurement)
Network Probe (UDP Echo + TCP) Port 9090
LEGEND
Critical (High Priority)
Best-Effort (Throttled)
Probe (Sensor)

Figure 4: Kubernetes Cluster Topology

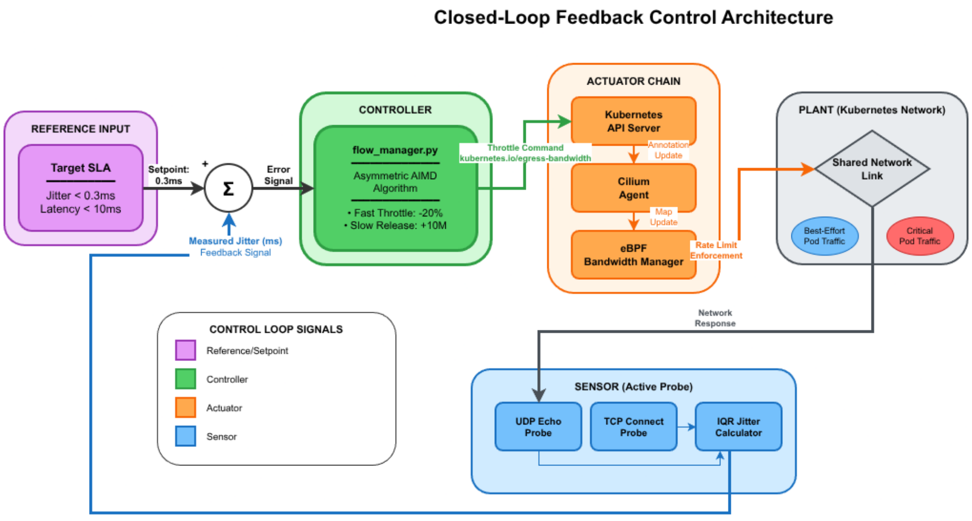


Figure 5: Closed-Loop Controller Architecture