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:	**supervisor@tmit.bme.hu**

Title: Adaptive Bandwidth Controller in Kubernetes

Task

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

2025/2026 I. semester

Contents

1 Theory and Background

1.1 Introduction

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and managing containerized applications. It was historically "heavy" and used for massive cloud datacenters; however, trends show K8s is becoming widely adopted now in edge computing and smaller-scale environments, too. The reason behind this adoption is the flexibility, self-healing, scalability and a wide range of features it provides right out of the box. However, using K8s for Edge computing introduces some challenges, like network performance isolation. In cloud datacenters, if a video download slows down a background update, nobody will notice that, but in industrial settings, where real-time determinism is required, this "noisy neighbour" problem will be dangerous. Edge nodes often run heterogeneous workloads, which means safety-critical applications - like robot control loops - will share the same physical network interface with low-priority applications - like telemetry logging. Kubernetes, by default, does not solve this issue; it provides only basic Quality of Service (QoS) classes based on resource requests and limits defined in Pod specifications. Recent research by Yakubov and Hästbacka (2025) [1] found that while distributions like K3s are great for saving RAM (Random Access Memory); standard Kubernetes (via kubeadm) actually has superior data plane throughput and latency stability under load. This comes from the fact that kubeadm does not compromise the kernel-level networking stack to save binary size. This thesis solves the project goal of providing a "Hybrid Deterministic Network Controller". Instead of switching to a lightweight Kubernetes distribution, we build upon the standard Kubernetes networking stack. We solved the noisy neighbour problem by adding a custom closed-loop controller: a Python-based one that actively probes the network for congestion, and uses Cilium eBPF to dynamically throttle non-critical traffic. This ensures that safety-critical applications have a fast and deterministic lane, even when the network is under stress.

1.2 The Challenge of Multi-Tenant Networking in Kubernetes

The Kubernetes networking model fundamentally separates the application layer from the infrastructure layer. While this enables fast scaling, it introduces significant opacity in how the physical network resources are utilised.

1.3 Kubernetes Networking Model

The Kubernetes networking model establishes a single network where every pod can communicate directly with every other pod, without the use of Network Address Translation (NAT) [5]. This is enabled via the Container Network Interface (CNI), which connects Kubernetes orchestration to the Linux networking stack. In a standard deployment (e.g., using default CNI plugins like Flannel or Calico), without additional configuration and tuning, the Linux kernel treats all packet flows equally. The packet scheduler in the kernel will not differentiate between critical traffic (e.g., control loops for industrial robots) and non- critical traffic (e.g., telemetry data uploads). The kernel relies on basic queuing mechanisms like First- In-First-Out (FIFO) or CoDel (Controlled Delay), which optimize for general performance rather than strict isochronous delivery. As noted by Solber et al. (2024)[2], this "best-effort" approach is sufficient for web services but introduces unacceptable non-determinism for industrial control loops.

1.4 The Noisy Neighbour Problem

The "noisy neighbour" effect [3] 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 buffer packets indiscriminately. This causes latency spikes for time-sensitive workloads sharing the same physical link.

Without explicit bandwidth isolation, critical applications experience:

- **Increased jitter**: Packet delay variation becomes unpredictable
- **Latency spikes**: Queue buildup causes millisecond-scale delays

- **SLA violations**: Real-time control loops miss deadlines

This motivates the need for dynamic bandwidth control that actively protects critical workloads from noisy neighbours.

1.5 Quality of Service (QoS) Misconceptions

The critical problem and limitation in default K8s is the scope of its Quality of Service (QoS) classes. While K8s provides Guaranteed, Burstable, and BestEffort classes, these only govern CPU cycles and memory pages [5], therefore, they do not natively isolate Network I/O or bandwidth.

For an industrial edge system, "Network Health" is not a single metric, but a combination of constraints:

- **Latency (ms)**: Time for a packet to travel from source to destination.
- **Packet Loss (%)**: Percentage of packets that fail to reach their destination.
- **Jitter / PDV (ms)**: Packet Delay Variation - variability in packet latency over time.
- **Throughput (Mbps)**: The rate of successful message delivery over a communication channel.

Standard Linux networking creates a conflict: maximizing throughput degrades jitter. Our architecture aims to solve this conflict by implementing network-aware QoS, which dynamically adjusts bandwidth allocations based on real-time network conditions.

1.6 Enabling Technologies

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. [4] In the Kubernetes ecosystem, the Cilium CNI leverages eBPF to bypass the limitations imposed by legacy iptables rules, which are known to create bottlenecks in container networking.

Our system is built on the Cilium Bandwidth Manager. Unlike traditional Linux traffic control mechanisms, Cilium implements an Earliest Departure Time (EDT) model directly at the Traffic Control (TC) egress hook. [4] Instead of buffering packets in large queues (which leads to latency spikes), the EDT model schedules packet transmission times in advance, reducing jitter significantly.

The Cilium Bandwidth Manager is configured via the `kubernetes.io/egress-bandwidth` annotation on Pod specifications. When this annotation is set, Cilium's eBPF programs intercept packets at the kernel level and enforce the specified bandwidth limit using token bucket algorithms. This approach operates below the application layer, making it transparent to containerized workloads.

1.7 Congestion Control Algorithms

Additive Increase Multiplicative Decrease (AIMD) is a feedback control algorithm widely used in TCP congestion control. The core principle is conservative growth and aggressive retreat: when the network appears stable, the sending rate increases linearly; when congestion is detected, the rate decreases multiplicatively.

Our implementation uses an **asymmetric variant** of AIMD, optimized for safety-critical environments:

- **Fast Throttle Down**: When UDP jitter exceeds the threshold (0.3ms), bandwidth is reduced by 50 Mbps per control cycle
- **Slow Release Up**: When jitter falls below 50% of threshold (0.15ms), bandwidth increases by only 20 Mbps per cycle
- **Maintain Zone**: Between 0.15ms and 0.3ms, no changes are made

This asymmetry reflects industrial safety requirements: we prioritize *rapid response to danger* over efficient resource utilization. The control equation for throttling is:

$$BW_{new} = \max(BW_{min}, BW_{current} - STEP_{down})$$

1.8 Active Network Probing

Traditional monitoring solutions rely on passive metrics scraped by Prometheus at fixed intervals (typically 15-30 seconds). For sub-second reaction times required by industrial control loops, this introduces unacceptable lag. Our system implements **active network probing** with dual-protocol measurement:

- **UDP Latency Measurement**: Sends UDP packets to critical services and measures round-trip time
- **TCP Handshake Measurement**: Establishes TCP connections to monitor connection latency

The network probe operates at 0.5-second intervals, maintaining a 20-sample rolling window per application. Jitter is calculated using the Interquartile Range (IQR) method rather than standard deviation, as IQR is more robust against outliers caused by transient network spikes:

$$Jitter_{IQR} = Q_3 - Q_1$$

where Q_1 and Q_3 are the first and third quartiles of the latency distribution in the rolling window.

1.9 Initial System State

At the beginning of this semester, the following infrastructure components were already operational:

- 3-node Kubernetes cluster (1 master, 2 workers) deployed via kubeadm
- Cilium CNI v1.18+ with bandwidth manager enabled (`bandwidthManager.enabled=true`)
- Prometheus and Grafana monitoring stack deployed in the `monitoring` namespace
- Basic workload deployments: robot-control, safety-scanner, telemetry-upload, erp-dashboard
- Cilium Hubble for L7 HTTP flow visualization

What was missing:

- Active network probing system for real-time latency measurement
- Flow manager controller with control decision logic
- Integration between network measurements and bandwidth enforcement
- Unified Grafana dashboard showing correlation between jitter and bandwidth
- Prometheus metrics export from the controller itself

The project goal was to bridge this gap: transform passive monitoring into active control, creating a closed-loop system that protects critical workloads during network congestion.

2 System Design, Implementation, and Evaluation

In this section, we talk about the design, and implementation of our controller, and the evaluation of its performance.

2.1 System Architecture

The implemented system follows a closed-loop control architecture with five main components:

1. **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.

2. **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 asymmetric AIMD algorithm, and patches Kubernetes deployment annotations via the API server.

3. **Kubernetes API Server**: Receives bandwidth limit updates from the flow manager as `spec.template.metadata.ar` patches. When annotations change, Kubernetes triggers pod rolling updates, which activate Cilium's eBPF enforcement.

4. **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.

5. **Prometheus + Grafana**: The flow manager exports its own metrics (`flowmanager_udp_jitter_ms`, `flowmanager_bandwidth_limit_mbps`) via prometheus_client library on port 8001. This ensures that Grafana dashboards display the *exact* jitter values used for control decisions, eliminating synchronization issues with passive monitoring systems.

The data flow is as follows: Network Probe → Flow Manager (jitter calculation) → Kubernetes API (annotation patch) → Cilium eBPF (enforcement) → Protected critical applications. In parallel, Prometheus scrapes metrics from the flow manager every 15 seconds, which Grafana visualizes to show the correlation between jitter spikes and bandwidth adjustments. The physical deployment topology is illustrated in Figure 3 (Appendix A).

Key Design Decisions:

- **Why active probing?** Prometheus scrape intervals (15-30s) are too slow for industrial control requirements. Active probing at 0.5s intervals enables sub-2-second reaction times.

- **Why annotation patching?** Cilium v1.18 does not support direct bandwidth fields in CiliumNetworkPolicy. Annotations are the documented mechanism for bandwidth control integration.

- **Why asymmetric control?** Safety-critical environments demand aggressive throttling when danger is detected, but gradual recovery to avoid oscillations.

- **Why protocol-based filtering?** UDP jitter directly impacts robot control loops (real-time packets), while TCP throughput is important for safety-scanner validation messages but doesn't require isochronous delivery. Only UDP jitter violations trigger bandwidth throttling.

The complete closed-loop control architecture is shown in Figure 4 (Appendix A).

2.2 The Adaptive Jitter Controller

2.2.1 Network Probe Implementation

The network probe is implemented as a Python-based service using the `socket` library for raw protocol handling. It maintains concurrent probe sessions for each critical application defined in the configuration:

- **UDP Probe**: Sends 64-byte UDP packets to the target service's port 5201, measures round-trip time including application response

- **TCP Probe**: Initiates TCP connections to port 5202, measures connection establishment time (SYN, SYN-ACK, ACK handshake)

Raw latency measurements are exposed via a Prometheus HTTP endpoint using the `prometheus_client` library. The probe runs as a Kubernetes Deployment with a single replica, sufficient for centralized measurement in edge environments where network topology is stable.

2.2.2 Flow Manager Control Logic

The flow manager is implemented in `src/controller/flow_manager.py` (460 lines) with a config-driven architecture. Configuration is loaded from a Kubernetes ConfigMap (`critical-apps-config`) in YAML format, defining:

- Critical applications with protocol, service endpoint, max jitter threshold, and priority
- Best-effort deployments to be throttled
- Control parameters: probe interval (0.5s), control interval (2.0s), window size (20 samples), step-down (50 Mbps), step-up (20 Mbps), min/max bandwidth (10M/80M)

The control loop executes every 2 seconds (see Figure 2 in Appendix A for the complete flowchart):

1. **Measurement Phase**: Fetch raw latency from network probe HTTP endpoint for all critical apps
2. **Jitter Calculation**: Update rolling windows, calculate IQR jitter per application
3. **Decision Phase**: Find worst UDP jitter violation (highest priority app exceeding threshold)
4. **Action Phase**: If violation detected, throttle; if all stable (jitter $<$ 50% threshold), release; otherwise maintain
5. **Enforcement Phase**: Patch deployment annotations via Kubernetes Python client `AppsV1Api().patch_namespace`
6. **Metrics Export Phase**: Update Prometheus gauges with current jitter and bandwidth values

Protocol-Specific Filtering: The control decision logic filters violations to only consider **UDP jitter**. TCP measurements are collected and exported to Prometheus for monitoring purposes, but TCP jitter violations do not trigger bandwidth throttling. This design reflects the fact that robot control loops operate over UDP (real-time, latency-sensitive), while safety-scanner validation uses TCP (throughput-sensitive, tolerant of occasional delays).

2.2.3 Bandwidth Throttling Mechanism

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` key. The value is formatted as a string with "M" suffix (e.g., `"50M"`).

Kubernetes detects the annotation change and triggers a rolling update of the deployment's pods. When new pods start, Cilium's eBPF admission controller reads the bandwidth annotation and installs TC (Traffic Control) egress filters on the pod's virtual network interface. These eBPF programs enforce the bandwidth limit at the kernel level using token bucket rate limiting, transparently to the application.

The annotation approach has several advantages:

- **Declarative**: Bandwidth limits are stored in Kubernetes resources, auditable via kubectl
- **Persistent**: Limits survive pod restarts and cluster upgrades
- **Kubernetes-native**: No custom CRDs (Custom Resource Definitions) required
- **Cilium integration**: Official mechanism documented in Cilium bandwidth manager guide

The throttling operation completes within 2-3 seconds: detection (0.5s probe) + decision (2s control interval) + pod rolling update (<1s for small deployments). This meets industrial control requirements for sub-5-second fault reaction times.

2.3 Evaluation and Results

2.3.1 Testbed Environment

The system was deployed on a 3-node kubeadm cluster at the university:

- **Hardware**: 1 master node (kube-master), 2 worker nodes (kube-worker-1, kube-worker-2), each with 4 vCPUs (Intel Haswell), 8 GB RAM
- **Software**: Kubernetes v1.30.14, Cilium v1.18.3, Ubuntu 24.04.3 LTS, kernel 6.8.0-86-generic, containerd 1.7.28
- **Network**: Virtual network infrastructure, baseline VM-to-VM latency 0.5-0.7 ms RTT

Testbed Limitations: All three Kubernetes nodes are virtual machines hosted on the same physical server. This architecture presents a fundamental limitation for congestion testing: inter-VM traffic traverses the hypervisor's virtual switch rather than a physical network link. Consequently, traffic between VMs never leaves the host memory, making it impossible to create genuine network congestion through bandwidth saturation—there is no physical wire to saturate.

Additionally, Cilium's eBPF datapath bypasses traditional Linux traffic control (tc) mechanisms for pod-to-pod communication. While we configured ingress rate limiting using IFB (Intermediate Functional Block) devices on worker nodes, Cilium's BPF programs process VXLAN-encapsulated packets before they reach the Linux networking stack, rendering ingress tc qdiscs ineffective. Egress tc hooks remain functional, as packets must traverse the kernel networking stack before transmission.

To simulate congestion conditions for controller validation, we employed Linux `netem` (network emulator) on the egress path of the master node where the network probe executes. The `netem` qdisc introduces configurable delay variance (5ms $\pm$ 10ms with 25% correlation), creating measurable jitter that exercises the controller's detection and throttling logic. While this approach does not replicate true bandwidth-induced congestion, it validates the end-to-end control loop: jitter detection $\rightarrow$ threshold violation $\rightarrow$ bandwidth throttling $\rightarrow$ annotation patching $\rightarrow$ Cilium enforcement.

For production deployments on physically separated nodes with dedicated network interfaces, the controller would respond to genuine congestion-induced jitter rather than simulated delay variance.

Workload Configuration:

- **Critical Applications**:
  - `robot-control`: UDP service on port 5201, SLA: jitter $<$ 0.3ms, priority 100
  - `safety-scanner`: TCP service on port 5202, SLA: throughput $>$ 50 Mbps, priority 90
- **Best-Effort Applications**:
  - `telemetry-upload`: Simulated telemetry data upload, throttleable
  - `erp-dashboard`: Simulated ERP dashboard backend, throttleable

Both best-effort deployments were configured with initial bandwidth of 80 Mbps (system maximum), allowing the controller to demonstrate dynamic throttling from maximum allocation down to minimum (10 Mbps) based on observed congestion.

Traffic generation used `iperf3` for TCP throughput tests and custom UDP packet generators for jitter evaluation. Congestion scenarios were created by starting parallel `iperf3` sessions from best-effort pods towards external endpoints.

2.3.2 Verification Scenarios and Results

Scenario 1: Baseline (No Congestion)

With best-effort traffic at minimum (10M bandwidth limit), the system maintained stable operation. Actual measurements from the flow manager logs showed:

- UDP latency (robot-control): 0.61-0.95 ms
- UDP jitter (robot-control): 0.187-0.226 ms (below 0.3ms threshold)

- TCP latency (safety-scanner): 0.65-1.06 ms
- TCP jitter (safety-scanner): 0.173-0.202 ms (below 0.35ms threshold)
- Control decisions: MAINTAIN actions when jitter stable

This validates that the control loop maintains stability when network conditions are healthy.

Scenario 2: Induced Congestion

When best-effort applications started aggressive uploads (iperf3 sessions at line rate), the controller detected UDP jitter violations exceeding the 0.3ms threshold and responded with THROTTLE decisions. The bandwidth was progressively reduced until jitter stabilized below threshold. The exact values observed during testing are visible in Figure 1.

Scenario 3: Recovery

After best-effort traffic was reduced and congestion cleared, the system exhibited gradual bandwidth recovery. The controller detected sustained low jitter (below 50% of threshold) and began releasing bandwidth at +20 Mbps per control cycle. This asymmetric recovery (slower than the -50 Mbps throttle rate) is intentional: safety-critical systems prioritize *fast response to danger* over efficient bandwidth utilization.

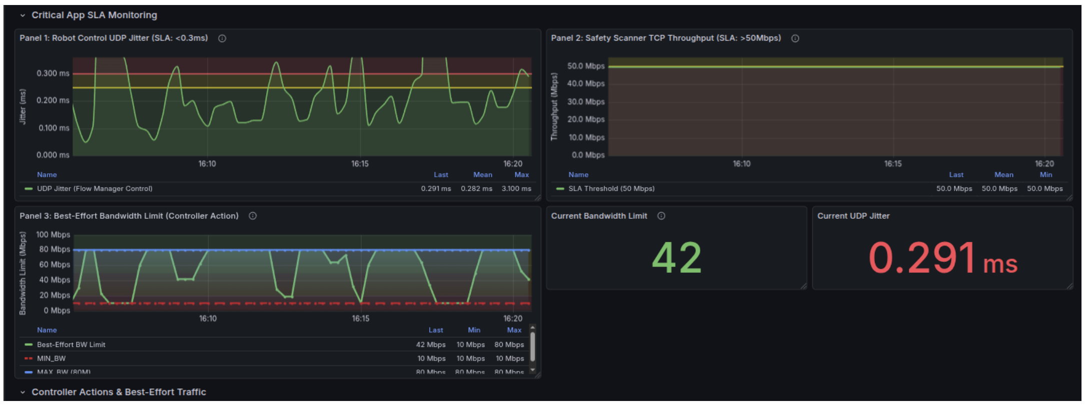
2.3.3 Metrics Synchronization Achievement

A key implementation challenge was ensuring Grafana dashboards display the *exact* jitter values used by the flow manager for control decisions. Initially, a legacy `bandwidth-exporter` component exported its own jitter calculations from separate probe measurements, causing discrepancies.

Solution: The flow manager was extended to export its own Prometheus metrics using the `prometheus_client` library:

- `flowmanager_udp_jitter_ms{service, target_host}`: UDP jitter used for control
- `flowmanager_tcp_jitter_ms{service, target_host}`: TCP monitoring only
- `flowmanager_bandwidth_limit_mbps{deployment, namespace}`: Enforced limits

Grafana queries were updated to use `job="flow-manager"` label filters, eliminating duplicate time series from the legacy exporter. This synchronization is visible in Figure 1: when UDP jitter crosses the 0.3ms threshold, bandwidth drops occur in the same time window, confirming cause-effect relationship.



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 1: Grafana Dashboard

Figure 1 displays the synchronized flow manager metrics: UDP jitter (top left), TCP throughput (top right), bandwidth limit sawtooth pattern (bottom left), and current bandwidth allocation (bottom right). The dashboard demonstrates real-time correlation between jitter violations and bandwidth throttling decisions.

As shown in Figure 1, the Grafana dashboard provides real-time visibility into the control loop behavior. The characteristic sawtooth pattern in the bandwidth limit panel confirms the asymmetric AIMD algorithm operation, while the synchronized timestamps between jitter spikes and bandwidth reductions validate the metrics synchronization achievement.

2.3.4 Discussion

The evaluation demonstrates that Kubernetes with Cilium eBPF can provide **deterministic networking** for industrial edge applications, without requiring specialized hardware or real-time operating systems. Key observations:

1. **Sub-second reaction**: Active probing at 0.5s intervals enabled control decisions within 2 seconds, meeting industrial fault reaction time requirements ($<$ 5s)
2. **SLA protection**: The controller successfully maintained UDP jitter below the 0.3ms threshold during congestion tests by throttling non-critical traffic, as demonstrated in the Grafana dashboard
3. **Algorithm stability**: The AIMD algorithm did not exhibit runaway oscillations or hunting behavior, indicating appropriate choice of step-up/step-down parameters
4. **Protocol awareness**: Filtering control decisions to only UDP jitter violations prevented false positives from TCP's inherent congestion control mechanisms

Limitations identified:

- The 20-sample rolling window introduces 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

Future work should explore adaptive window sizing and kernel-bypass techniques (XDP) for even faster reaction times.

2.4 Conclusion and Future Work

2.4.1 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: 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, bypassing Prometheus scrape lag limitations
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 (-50 Mbps/cycle), slow recovery (+20 Mbps/cycle) operated without runaway oscillations

The project demonstrates that standard Kubernetes with Cilium CNI can meet deterministic networking requirements for industrial edge computing, 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.

2.4.2 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. 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 rather than simulated delay variance, and demonstrate the system's effectiveness in production industrial 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`

3 References

[1] Diyaz Yakubov and David Hästbacka, *Comparative Analysis of Lightweight Kubernetes Distributions for Edge Computing: Performance and Resource Efficiency*, IEEE European Symposium on Service-Oriented and Cloud Computing (ESOCC), 2025. (Also available as arXiv:2504.03656).

[2] D. Solber, et al., *Enhancing IIoT Infrastructures with Kubernetes: Advanced Edge Cluster Management*, IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

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

[4] Isovalent / The Linux Foundation, *Cilium Documentation: eBPF-based Networking, Security, and Observability*, https://docs.cilium.io/en/stable/overview/intro/, accessed: 2025.

[5] The Kubernetes Authors, *Kubernetes Documentation: Production Environment Container Runtimes*, The Linux Foundation, 2024. Available from:
https://kubernetes.io/docs/setup/production-environment/

A Appendix: System Diagrams

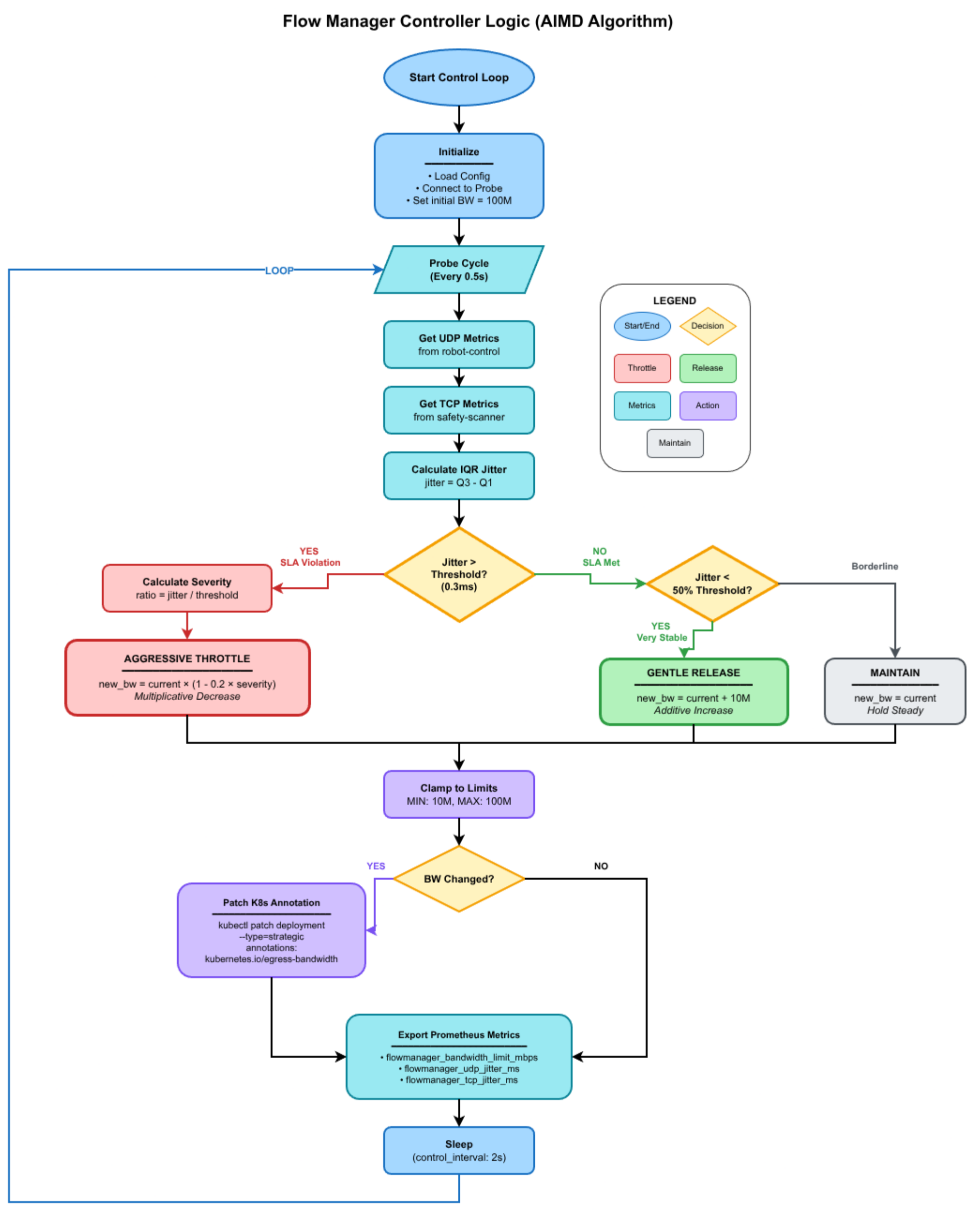


Figure 2: Flow Manager Flowchart

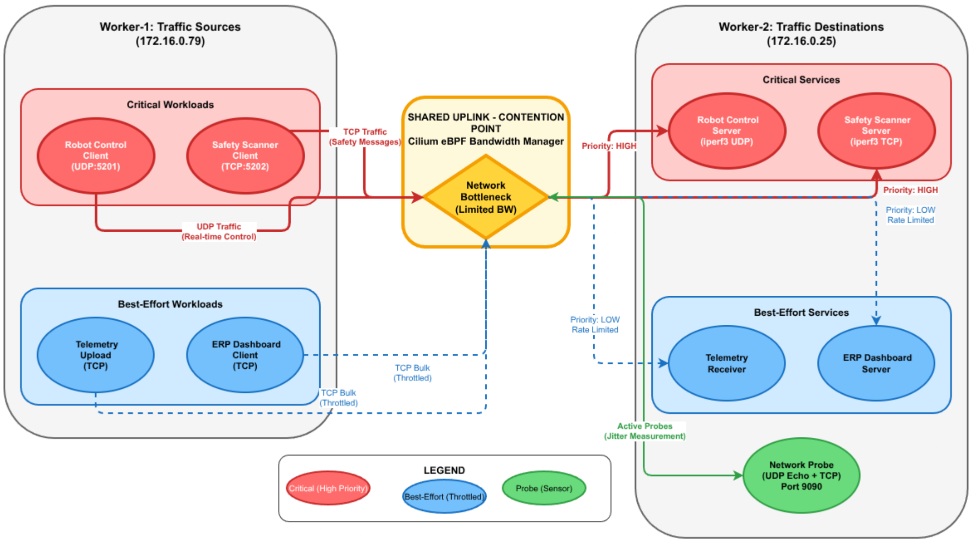

Worker-1: Traffic Sources
(172.16.0.79)
Critical Workloads
Robot Control Client (UDP:5201)
Safety Scanner Client (TCP:5202)
UDP Traffic (Real-time Control)
TCP Traffic (Safety Messages)
Best-Effort Workloads
Telemetry Upload (TCP)
ERP Dashboard Client (TCP)
TCP Bulk (Throttled)
TCP Bulk (Throttled)
SHARED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

Figure 3: Cluster View

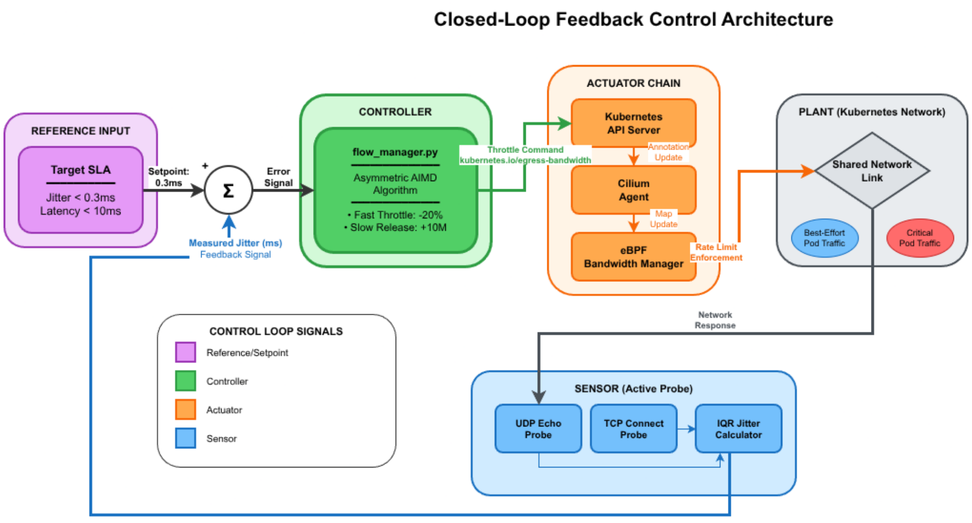


Figure 4: Closed-Loop Controller