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

Short description of the project.

2025/2026 I. semester

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 provides that the 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) [4]. 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 won't 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), 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 Kubernetes networking model enforces a "flat" network space where every pod can communicate with every other pod without NAT (Network Address Translation) [4]. This is achieved through the Container. Network Interface (CNI), which acts as the translation layer between the K8S orchestration and the underlying Linux networking stack. In a standard deployment (e.g., using default CNI plugins like Flannel or Calico), without additional configuration or tuning, the Linux kernel treats all packet flows equally. The packet scheduler in the kernel won't 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 FIFO (First-In-First-Out. In First Out (FIFO) or CoDel (Controlled Delay), optimise 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.

1.5 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 [4], therefore, they do not natively isolate Network I/O or bandwidth.

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

- Latency(ms): Time to travel from source to destination.
- Packet Loss(percentage): Percentage of packets that fail to reach their destination.
- Jitter(PDV)(ms): 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. [3] 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- anisms, Cilium implements an Earliest Departure Time (EDT) model directly at the Traffic Control (TC) egress hook. [3] Instead of buffering packets in large queues (which leads to latency spikes). Something in here about something

2 System Design, Implementation, and Evaluation

2.1 System Architecture

2.2 The Adaptive Jitter Controller

2.3 Evaluation and Results

2.4 Conclusion and Future Work

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

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

3.1 Attached documents

Source code etc.