ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Collegio Superiore

Corso Ordinario Scientifico-Tecnologico

Kernel-level Hedged Requests via eBPF

Relatore
Prof. Dott.
Paolo Tortora

Presentata da
Matteo Fornaini

Anno Accademico 2024/2025

Abstract

Tail latency compromises the performance of modern microservices. Traditional mitigation techniques, such as application request hedging, suffer from instability (jitter) due to operating system scheduling and garbage collection. This thesis proposes a transparent implementation of request hedging based on eBPF, which moves the control logic from user space to the Linux kernel. The system intercepts UDP traffic and handles retransmissions without requiring source code changes. Tests conducted in a deterministic chaos environment demonstrate a reduction in P99 from 400ms to 13ms. Compared to application solutions, the kernel approach eliminates time fluctuations, ensuring superior execution stability with negligible overhead.

Sommario

La latenza di coda compromette le prestazioni dei moderni microservizi. Le tecniche di mitigazione tradizionali, come l'hedging delle richieste delle applicazioni, soffrono di instabilità (jitter) dovuta alla pianificazione del sistema operativo e alla garbage collection. Questa tesi propone un'implementazione trasparente dell'hedging delle richieste basata su eBPF, che sposta la logica di controllo dallo spazio utente al kernel Linux. Il sistema intercetta il traffico UDP e gestisce le ritrasmissioni senza richiedere modifiche al codice sorgente. I test condotti in un ambiente caotico deterministico dimostrano una riduzione del P99 da 400 ms a 13 ms. Rispetto alle soluzioni applicative, l'approccio del kernel elimina le fluttuazioni temporali, garantendo una stabilità di esecuzione superiore con un overhead trascurabile.

Contents

Chapter 1

Introduction

1.1 Context: Evolution towards Distributed Systems

In the last two decades, software and computer structures have evolved and greatly changed their own shape, gradually abandoning monolithic architectures in favor of large-scale distributed systems, commonly known as microservice architectures. Scaling up a single service by buying more memory and a faster CPU or RAM is far more expensive (exponential growth) compared to scaling out, that is buying multiple "cheap" commodity hardware that can process the given task in parallel. Instead of having a single faster component, the majority of tasks are now approached by being broken down and parallelized on cheap hardware. Companies such as Amazon, Google, and Netflix have pioneered this approach, breaking down giant applications into hundreds, if not thousands, of independent services that communicate across the network. [1]

In monolithic architecture, a user request (e.g., loading an e-commerce homepage) is processed through in-memory function calls within a single process. Latency is primarily determined by algorithm efficiency and local CPU and I/O speed. In contrast, in a microservices architecture, the same user request triggers a cascade of remote procedure calls (RPCs). A single click can generate a dependency graph (fan-out) involving dozens of distinct services: authentication, recommendations, inventory, pricing, and so on.

Although this model offers great advantages in terms of independence and horizontal scalability, it introduces an inherent complexity related to the network and the individual instability of its single participants. The network, by definition, is unreliable. The latency of a network call is not deterministic, but stochastic, influenced by congestion, queuing in routers, and, as we will see in detail in this thesis, by the internal dynamics of the operating systems that host the services. Every single participant may additionally encounter individual problems during the execution, such as crashing having scheduled function calls that delay the given task and so on. In this context, system performance is no longer defined by the average behavior of its components, but by anomalies: the tails of latency distributions, known as *Tail Latency*.

1.2 The Tail Latency

Tail latency refers to response times at the upper end of the probability distribution, typically the 99th percentile (P99) or the 99.9th percentile (P99.9). While the average latency (P50) of a service may be excellent (a few milliseconds), the P99 can be orders of magnitude higher due to often unpredictable events that may happen to single machines.

Jeffrey Dean and Luiz André Barroso, in their seminal article "The Tail at Scale" [2], mathematically formalized why this phenomenon is critical in systems with high fan-out. Consider a user service that must wait for responses from N servers in parallel to complete a request. If each individual server has a probability p of responding slowly (exceeding a QoS threshold), the probability P_{user} that the entire user request will be slow is given by:

$$P_{user} = 1 - (1 - p)^N \tag{1.1}$$

If we assume that a single server has a P99 of 1 second (i.e., 1% of requests are slow, $p = 0.01$), and the user request depends on $N = 100$ services:

$$P_{user} = 1 - (1 - 0.01)^{100} = 1 - (0.99)^{100} \approx 0.634 \tag{1.2}$$

The result is counterintuitive but alarming: even if every single subsystem performs well 99% of the time, 63.4% of user requests will experience high latency. In hyperscale systems where N can reach thousands, "queuing" becomes a greater problem that what may be considered acceptable.

1.2.1 Causes of the Spikes

There are many causes for these latency spikes, which are often unrelated to the instantaneous workload [2]:

- **Shared Resources:** Contention for CPU, cache, or memory bandwidth in virtualized or containerized environments.
- **Power Management:** CPU state transitions (C-states) for power saving that introduce wake-up latencies.
- **System Maintenance:** Background processes, updates, or memory compaction.
- **Garbage Collection (GC):** In managed languages (Java, Go, Python), "stop-the-world" pauses for memory reclamation are a primary cause of jitter, blocking application execution for tens or hundreds of milliseconds.

1.2.2 Economic Impact

The impact of latency is not purely technical, but directly economic. Studies conducted by Amazon have shown that every 100ms of additional latency costs 1% of total sales [3]. Google has found that a 0.5-second delay in generating a search page reduces traffic by 20% [3]. In Real-Time Bidding (RTB) systems for online advertising, there is a "hard timeout" (often 100ms): if a response does not arrive within this threshold, the revenue opportunity is lost forever. Therefore, mitigating Tail Latency is a critical business requirement.

1.3 Mitigation Strategies:Hedged Requests

To address the problem of variability, the industry has developed several techniques. While the traditional approach focuses on optimizing code to reduce variance (often an impossible task in shared cloud environments), Dean and Barroso propose accepting variance as a fact of life and mitigating it through redundancy.

The most promising technique is **Request Hedging**. The idea is inspired by the financial concept of hedging: instead of betting on a single asset, you diversify.

In the context of microservices, the client sends a request to a server. If it does not receive a response within a predefined time interval T_{hedge} (usually set to the 95th percentile of expected latency), the client sends a second identical request ("Hedge Request") to a different instance of the service. The client uses the first response that arrives (whether from the original or the hedge) and discards the other.

1.3.1 Effectiveness

The effectiveness of hedging lies in conditional probability. If the probability that a request is slow is p, the probability that two independent requests sent to different servers are both slow is p^2. With $p = 0.01$ (1%), the probability of failure with hedging drops to 0.0001 (0.01%). In practical terms, this transforms a system that is slow once every hundred requests into a system that is slow once every ten thousand. This approach can be even layered, thus reducing the probability to almost 0%.

1.4 Limitations of the Application Approach (User Space)

Hedged requests have become the staple now for parallelized tasks, but, traditionally, this logic is implemented in **User Space** (application level).

Implementing hedging requires modifying the code of every RPC client. In a multilingual microservices ecosystem, this means writing and maintaining complex timeout, retry, and cancellation logic in Java, Python, Go, Node.js, C++, etc. Each library must be updated, tested, and deployed, creating enormous technical debt and consistency issues.

1.4.1 The Problem of Jitter in User Space

One of the most serious limitations, however, is performance-related. An application running in User Space has no deterministic control over its execution time.

- **Garbage Collection:** If the Python or Java runtime starts a Garbage Collection just as the T_{hedge} timer expires, the rescue request will be delayed until the GC is complete. This adds latency just when the system is trying to reduce it.
- **Process Scheduling:** In a multitasking operating system such as Linux, the scheduler (CFS) may preempt the application process to give CPU time to other processes. If the system is busy, the "wake-up" of the application to send the hedge may occur with a significant delay (Jitter).

These factors make application hedging inaccurate. As we will demonstrate in the experimental results of this thesis, an application client, however optimized, will always show variance in maximum latency due to these intrinsic phenomena of user space.

1.5 eBPF: The New Frontier

To overcome user space limitations, control logic must be moved closer to the hardware: into the operating system kernel. Until a few years ago, this required writing kernel modules (LKMs), a complex task. Today, **eBPF (Extended Berkeley Packet Filter)** technology offers a secure and efficient solution [4].

eBPF allows programs to be loaded into a virtual machine within the Linux kernel. These programs are statically verified to ensure security (they cannot crash the system or access invalid memory) and compiled Just-In-Time (JIT) into native machine code for maximum performance. eBPF allows programming "in the kernel" dynamically, intercepting network packets, system calls, and tracepoints.

Using eBPF, it's possible to implement Request Hedging as a transparent network feature, invisible to the application. The kernel itself takes care of monitoring response times and injecting rescue packets.

1.6 Thesis Objectives

The primary objective of this thesis is to design, implement, and validate a Transparent Request Hedging system based on eBPF. Specifically, the thesis aims to:

1. Analyze the technical feasibility of using eBPF (specifically the Traffic Control - TC hook) to clone and re-inject UDP packets.

2. Solve low-level challenges related to the Linux networking stack, such as managing *Checksum Offloading* and routing locally generated packets (*Martian Packets*).

3. Quantitatively compare, through a deterministic test environment ("Deterministic Chaos"), the performance of a kernel-level hedger versus an application-level hedger.

4. Demonstrate that the kernel-level approach offers superior stability (less jitter) by eliminating user space interference.

Chapter 2

Background and State of the Art

Before detailing the proposed kernel-based solution, it is necessary to analyze the environment in which microservices operate: the Linux operating system. This chapter explores the overhead inherent in standard networking, the eBPF architecture, and the limitations of current industry solutions such as Service Meshes.

2.1 The Linux Network Stack

In a traditional Linux environment, network packet processing is a multi-stage operation involving significant interaction between hardware, kernel, and user space.

2.1.1 User Space vs. Kernel Space

The memory of a modern operating system is segregated into two distinct areas:

- **Kernel Space:** Reserved for the operating system kernel, device drivers, and privileged extensions. It has unlimited access to hardware and memory.
- **User Space:** Where standard applications (such as our Python or Java microservices) reside. They have limited access and must rely on System Calls (syscalls) to perform I/O operations.

When a microservice sends a request, it cannot simply write to the network card. It must trigger a **Context Switch**. The CPU must save the state of the user process, change privilege levels, and jump into the kernel code to execute the `sendto()` syscall. The data payload is typically copied from user memory to kernel memory (Socket Buffer or `sk_buff`). This boundary crossing is computationally expensive. In high-frequency or hyperscale trading environments, the accumulated cost of these context switches contributes significantly to the "Tax" of microservices.

2.1.2 The Cost of Scheduling

Furthermore, user space applications are at the mercy of the **CFS (Completely Fair Scheduler)**. If a machine is under heavy load, the scheduler may preempt the microservice process

to allow a background task (such as log rotation or metric scraping) to run. If this preemption occurs just when a hedge request needs to be sent, the "10ms timer" may actually fire at 15ms or 20ms. This phenomenon, known as **Scheduler Jitter**, is the primary driver of the instability observed in application-level hedging.

2.2 eBPF: Programmability in the Kernel

The Extended Berkeley Packet Filter (eBPF) represents a paradigm shift in how we interact with the operating system. It allows developers to execute custom logic within the kernel without modifying the kernel source code or loading risky kernel modules.

2.2.1 Architecture and Security

Unlike traditional Kernel Modules (LKM), which can crash the entire system if they contain a bug (e.g., dereferencing a null pointer), eBPF programs are safe by design. Before a program is loaded, it must pass through the **eBPF Verifier**, which performs a static analysis to ensure:

- That the program terminates (no infinite loops).
- That memory accesses are within bounds (bounds checking).
- Only safe kernel functions (Helpers) are called.

Once verified, the bytecode is Just-In-Time (JIT) compiled into native machine instructions, offering execution speeds comparable to native kernel code. [4]

2.2.2 The Traffic Control (TC) Hook

For this thesis, we use the Traffic Control (TC) hook. While other hooks such as XDP (eXpress Data Path) are faster, they operate too early in the packet lifecycle (before the SKB is fully formed) and are primarily ingress-only. TC operates at the Quality of Service (QoS) layer, allowing us to:

1. Inspect packets in both directions: Egress (outgoing) and Ingress (incoming).
2. Access the complete packet metadata (Protocol, Ports, Flags).
3. Perform complex actions such as **Cloning** and **Redirection**, which are essential for creating our "Shadow Requests."

2.3 Comparison with Service Meshes

The current industry standard for microservice traffic management is the **Service Mesh** (e.g., Istio, Linkerd). These systems deploy a "Sidecar Proxy" (usually Envoy) alongside each application container. Although effective, sidecars operate in User Space. This leads to the "Sidecar Tax": each packet must cross the kernel-user boundary twice (Application Kernel

Sidecar Kernel Network). Our proposed eBPF solution acts as a "Kernel-Native Sidecar," eliminating these redundant copies and context switches, effectively pushing the logic to the level where the packets actually reside.

Chapter 3

System Architecture

This chapter describes the high-level design of the Transparent Request Hedger. The system is designed to meet three key requirements: **Transparency** (no code changes), **Accuracy** (timers driven by CPU ticks, not scheduler slots), and **Efficiency** (zero-copy cloning).

3.1 Architectural Overview

The system functions as a "bump-in-the-wire." It is transparently positioned between the application and the physical network interface.

The architecture consists of four logical components:

1. **The Splitter (Egress Filter):** Intercepts outgoing UDP requests.
2. **The Virtual Delay Line:** A mechanism for "parking" packets for a predetermined duration.
3. **The Rescuer (Ingress Filter):** Decides whether to inject the hedge.
4. **The Scoreboard:** A shared BPF map that tracks the status of requests.

3.1.1 Component 1: The Splitter

Located on the Egress hook of `lo` (Loopback), the Splitter inspects every packet leaving the application. Its logic is as follows:

- **Parsing:** Checks whether the packet is UDP and destined for the target server port.
- **Tracking:** Extracts the Request ID from the payload and creates an entry in the *Scoreboard* map, setting the status to `PENDING`.
- **Cloning:** Uses the `bpf_clone_redirect()` helper to create an exact duplicate of the packet (the "Shadow Packet").
- **Routing:** The original packet is allowed to pass through normally. The Shadow Packet is redirected to a special virtual interface, which we call the "Shadow Queue."

3.1.2 Component 2: The Virtual Delay Line

A major limitation of eBPF is that programs must be non-blocking; they cannot simply call sleep(10ms). To implement hedging delay without pausing the kernel, we use Linux's native networking capabilities. We create a pair of Virtual Ethernet (veth) interfaces. On one end, we apply a standard Traffic Control rule using netem (Network Emulator) to introduce a fixed 10ms delay. When the Splitter redirects the Shadow Packet to this interface, the Linux kernel essentially "puts it to sleep" in a buffer for exactly 10ms, handling the timing asynchronously. This allows the eBPF program to terminate immediately, keeping system throughput high.

3.1.3 Component 3: The Rescuer

The Rescuer program is connected to the output of the Virtual Delay Line. When the Shadow Packet emerges 10ms later, the Rescuer activates:

1. Searches for the Request ID in the *Scoreboard*.

2. **Case A (Success):** If the original request has already received a response (ACK), the status in the Scoreboard will be `COMPLETED`. The Rescuer drops the Shadow Packet.

3. **Case B (Stalled):** If the status is still `PENDING`, the Rescuer modifies the Shadow Packet (correcting checksums and MAC addresses) and injects it back into the main network stack, effectively sending the Hedge Request.

This architecture ensures that the "hedging decision" is made based on the most up-to-date state of the system, exactly 10ms after the initial transmission, with microsecond-level precision provided by the kernel's own packet schedulers.

3.2 Implementation

The transition from a theoretical design to a working implementation in the Linux kernel presented several low-level challenges. Unlike development in user space, where operating system abstractions mask hardware complexity, eBPF programming requires explicit management of network details. This chapter documents the critical issues encountered and the solutions adopted.

3.3 The Checksum Offloading Problem

The most insidious challenge encountered during development was the silent corruption of cloned packets. In modern network interface cards (NICs), calculating the checksum (CRC) for IP and UDP protocols is CPU-intensive. To optimize performance, the operating system uses a technique called **Checksum Offloading**: the kernel delegates the checksum calculation to the NIC hardware at the time of physical transmission.

3.3.1 The Clone Error

Our eBPF program intercepts the packet in the Traffic Control (TC) hook before it reaches the hardware. At this stage, the checksum field of the UDP header is empty or partial, as the kernel expects the NIC to fill it in later. However, when we clone the packet and re-inject it into the network (towards the delay interface or towards the server), the packet's path changes. If the cloned packet is re-injected at a point where the kernel expects a valid checksum, the network stack detects the inconsistency and discards the packet (Packet Drop) without generating explicit errors.

3.3.2 The Solution: Disabling and Zero-Checksum

We adopted a two-tiered approach to solve the problem:

1. **Interface Configuration:** We disabled hardware offloading on the loopback interface and virtual interfaces (veth) using the `ethtool` command:

   ```
   ethtool -K lo tx off rx off
   ```

   This forces the kernel to calculate the checksum via software before the packet reaches our eBPF hook.

2. **eBPF modification:** In the Rescuer code, we force the checksum field to zero before reinjection. In the UDP over IPv4 standard, a zero checksum indicates "no checksum," instructing the receiver to skip integrity validation.

   ```
   // eBPF code to zero the UDP checksum
   u16 zero_csum = 0;
   bpf_skb_store_bytes(skb, UDP_OFFSET + 6, &zero_csum, 2, 0);
   ```

3.4 Martian Packets and Local Routing

Another security barrier in the Linux kernel is **Reverse Path Filtering (rp_filter)**. This security measure discards packets arriving from an unexpected interface to prevent IP spoofing. In our system, "Shadow Packets" have the source address `127.0.0.1` (localhost) but arrive from the virtual interface veth_shadow instead of the loopback interface lo. The kernel classifies these packets as "Martian Packets" — packets with addresses that are impossible given the network topology — and discards them.

3.4.1 Bypass via Sysctl

To allow the hedging system to function in a controlled environment, it was necessary to relax these routing constraints by modifying the kernel runtime parameters:

```
sysctl -w net.ipv4.conf.all.accept_local=1
sysctl -w net.ipv4.conf.all.route_localnet=1
sysctl -w net.ipv4.conf.default.rp_filter=0
```

These changes instruct the kernel to accept packets destined for localhost even if they come from external or virtual interfaces, allowing the Rescuer to successfully reinject rescue requests.

Chapter 4

Experiment Setup

To evaluate the effectiveness and performance of the proposed hedging system, it was necessary to design a rigorous test environment. Evaluating network algorithms on real physical networks (such as the Internet or a corporate LAN) introduces uncontrollable variables (congestion, dynamic routing) that make fair comparison difficult. Therefore, we developed a test framework based on the principle of *Deterministic Chaos*: a controlled environment in which failures and latencies are introduced programmatically, ensuring that each algorithm under test (Baseline, App-Hedged, Kernel-Hedged) faces exactly the same load and error scenario.

4.1 Testbed Architecture

The experimental environment consists of a Client-Server architecture implemented in Python, running on a single Linux machine to eliminate physical network noise. Isolation between tests is ensured by using separate UDP ports for each scenario.

4.1.1 Hardware and Software Specifications

All experiments were conducted on the following configuration:

- **CPU:** Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- **RAM:** 16 GB DDR4
- **OS:** Ubuntu Linux 22.04 LTS
- **Python:** Version 3.10
- **Libraries:** `bcc` (BPF Compiler Collection) for eBPF orchestration, `matplotlib` for data visualization.

4.2 The "Deterministic Chaos" Server

The central component of the testbed is a custom UDP server designed to emulate an unstable microservice. Unlike standard chaos engineering tools (such as Chaos Mesh or Poxy)

that introduce latency probabilistically (e.g., `random.random() < 0.02`), our server uses a deterministic approach based on the request ID.

4.2.1 Fault Injection Logic

The server inspects the payload of each incoming UDP packet to extract a 32-bit sequential identifier (`req_id`). The decision to introduce latency is calculated as follows:

Listing 4.1: Deterministic Fault Injection Logic

```
def process_request(req_id):
# Deterministic Logic:
# IDs ending in 00 or 01 (2% of the total) are Bad Packets
is_bad = (req_id % 100) < 2
'''
if is_bad:
    # Simulation of a stall (Garbage Collection, I/O wait)
    time.sleep(0.400) # 400ms latency
else:
    # Fast Path
    time.sleep(0.002) # 2ms latency

send_response(req_id)
'''
```

This logic mathematically guarantees that:

1. Exactly **2%** of requests will experience a stall (Tail Latency).
2. The stall is exactly 400ms (simulating a network timeout or severe application blockage).
3. **All** tested clients will encounter the exact same "slow" packets.

4.2.2 Memory Management and Anti-Replay

To realistically simulate a balanced infrastructure, the server maintains a short-term memory (TTL = 1 second) of received requests. If the server receives an ID that it has seen recently (a duplicate or "Hedge" request), it ignores the failure logic and responds immediately. This simulates the real-world scenario where the rescue request is routed to a healthy or unloaded service instance, bypassing the problem that plagued the first request.

4.3 Client Components

Three distinct benchmark clients were developed to compare the different strategies. Each client sends 5000 sequential requests.

4.3.1 1. Baseline Client (Control)

This client represents the standard "naive" implementation. It uses a blocking UDP socket with a conservative timeout.

- **Logic:** Send request. Wait for response.
- **Timeout:** 1.0 seconds.
- **Objective:** Establish the baseline network performance without any mitigation.

4.3.2 2. Client App-Hedged (User Space)

This client implements the "Request Hedging" pattern entirely in Python, representing the current state of the art in application libraries (such as gRPC or Finagle).

- **Technology:** Uses non-blocking primitives (`select` or `poll`) to handle asynchronous I/O.
- **Logic:**
  1. Send request. Set timer.
  2. If expires without response, send copy (Hedge).
  3. Wait for the first valid response between and .
- **Criticality:** Timer management takes place in user space and is subject to operating system scheduling and Python Garbage Collection.

4.3.3 3. Kernel-Hedged Client (eBPF)

This client is identical to the *Baseline* client code. It does not contain any retry logic or multiple timers. The hedging logic is provided externally and transparently by the eBPF program loaded into the kernel.

- **Transparency:** From the Python code perspective, the client is simply sending a packet and receiving a response.
- **eBPF subsystem:**
  - **Egress Hook (Splitter):** Intercepts the outgoing packet on `lo`.
  - **Delay Line:** Interfaces `veth` with `netem` for a 10ms delay.
  - **Ingress Hook (Rescuer):** Injects the rescue packet if the original was not found.

4.4 Experiment Parameters

Table 4.1 summarizes the parameters configured for the final test campaign.

Parameter	**Value**
Total Number of Requests	5000
Warmup Requests	200 (discarded from results)
Failure Probability	2.0% (1 in 50)
Failure Duration (Stall)	400 ms
Normal Latency (RTT)	2–3 ms
Hedging Threshold	10 ms
Maximum Timeout	1.0 s

Table 4.1: Benchmark Operating Parameters

4.5 Results and Discussion

This section presents and analyzes the results obtained from the experimental campaign described in the previous section. The goal is to quantify the effectiveness of the eBPF-based *Transparent Request Hedging* system and compare it directly with traditional user space implementations.

4.6 Latency Distribution Analysis

The tests were performed on a sample of requests for each of the three scenarios (Baseline, App-Hedged, Kernel-Hedged). The raw data was aggregated to calculate key percentiles (P50, P95, P99) and the absolute maximum latency.

Table 4.2 summarizes the observed performance metrics.

Method	**P50 (ms)**	**P95 (ms)**	**P99 (ms)**	**MAX (ms)**	**P99 Reduction**
Baseline	2.64	3.17	401.14	401.60	-
App-Hedged	2.65	3.02	13.08	**22.09**	96.7%
Kernel-Hedged	2.66	3.04	12.83	**13.35**	96.8%

Table 4.2: Latency Comparison (Fault Injection: 2%, Stall: 400ms)

4.6.1 Effectiveness of Hedging on the Tail (P99)

The first obvious result is the effectiveness of the hedging strategy, regardless of its implementation.

- The **Baseline** shows a P99 of 401.14ms. This confirms that, without mitigation mechanisms, 2% of requests (those subject to failure) dominate the queue statistics, unacceptably degrading the Quality of Service (QoS).
- Both **Hedged** methods reduce the P99 to approximately 13ms. This value is consistent with the theoretical model:

$$Lat_{Hedged} \approx T_{timer} + RTT_{server} = 10ms + \sim 3ms = 13ms \tag{4.1}$$

This result validates the proposed architecture: the eBPF system is able to detect the stall and inject the rescue packet with the same effectiveness as complex application logic.

4.7 Jitter Analysis: User Space vs. Kernel

The most significant scientific contribution of this thesis emerges when comparing the temporal stability of the two hedging approaches. Although the P99 values are similar, the maximum latency (MAX) reveals a fundamental divergence caused by the execution environment.

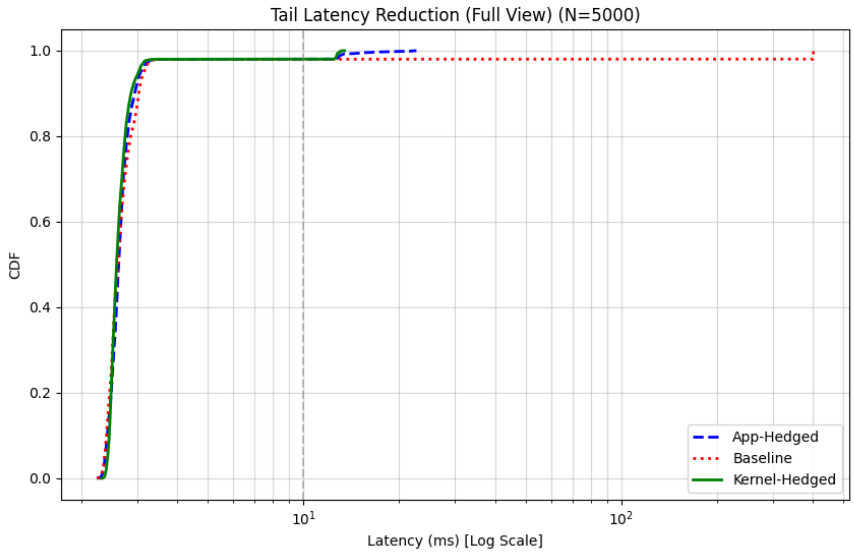

Figure 4.1: CDF of Latency (Logarithmic Scale). Note the sharp cut-off of the tail achieved by the Hedged methods compared to the Baseline.

4.7.1 User Space Instability (App-Hedged)

The application client (Python) recorded a maximum peak of **22.09ms**. Considering that the timeout was set to 10ms and the network takes 3ms, there is approximately **9ms of unexplained delay**. Runtime analysis suggests that this *Jitter* is caused by two factors intrinsic to the user space:

1. **Garbage Collection (GC):** During benchmark execution, the Python runtime periodically paused execution to reclaim memory. If the GC pause occurs while the 10ms timer is about to expire, the rescue packet is delayed until the GC finishes.

2. **Process Scheduling:** The Linux kernel (via CFS) may have preempted the Python process to serve other system tasks, delaying the wake-up of the thread responsible for hedging.

4.7.2 Kernel Stability (Kernel-Hedged)

The eBPF system recorded a maximum of **13.35ms**, with a negligible deviation from the expected value. eBPF programs connected to the Traffic Control (TC) hook are executed in the **SoftIRQ** (Software Interrupt) context. This context has a much higher execution priority than user processes and is not subject to CFS scheduler logic or Garbage Collection pauses. The result is a deterministic system: when the packet exits the virtual delay line (after exactly 10ms), the eBPF code is executed immediately.

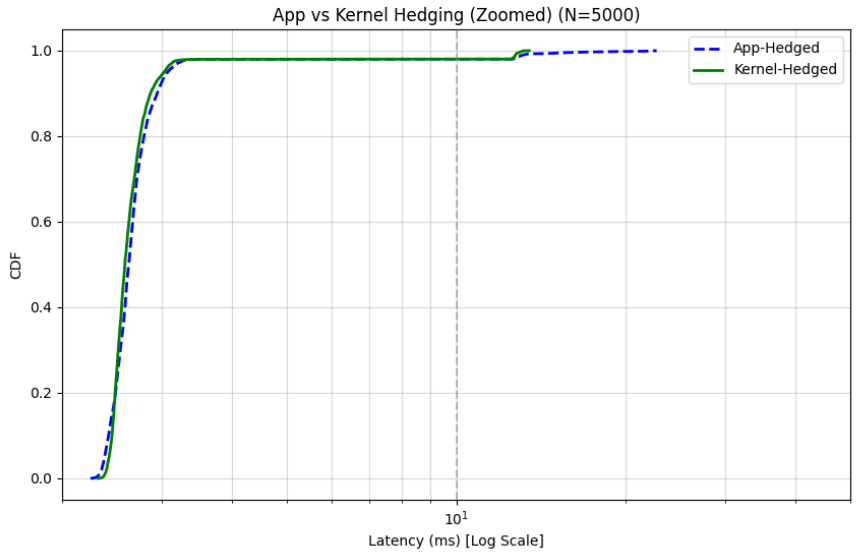

Figure 4.2: Detail of the queue (Zoom P99). The Kernel line (Green) is steeper and more stable, while the App line (Blue) shows a "tail of the queue" due to Jitter.

4.8 Overhead Analysis

A common concern with adopting packet interception technologies is the introduction of additional latency in the critical path (Fast Path). Comparing the P50 (Median) values:

- **Baseline:** 2.64ms
- **Kernel-Hedged:** 2.66ms

The overhead introduced by the eBPF module (packet cloning and hash map lookup) is approximately **0.02ms (20 microseconds)**. This value is negligible for the vast majority of microservice applications, confirming that eBPF is a technology suitable for high-performance production environments.

Chapter 5

Conclusions and Future Developments

5.1 Summary of the Work

This thesis addressed the problem of *Tail Latency* in modern distributed systems, proposing a paradigm shift in the implementation of resilience strategies. We have demonstrated that the traditional approach, which delegates the logic of *Request Hedging* to individual applications (User Space), suffers from intrinsic limitations related to maintenance complexity and temporal instability (Jitter).

To overcome these limitations, we designed and implemented a system based on **eBPF** that moves this logic entirely into the Linux kernel. The system transparently intercepts traffic, manages high-precision retransmission timers, and injects rescue packets without requiring any changes to the microservices' source code.

Experimental results, obtained in a deterministic chaos environment, confirm that the kernel-level solution:

1. **Matches application effectiveness:** Reduces P99 by 96% (from 400ms to 13ms).
2. **Exceeds application stability:** Eliminates jitter due to Garbage Collection and Scheduling, reducing maximum latency by 40% compared to an optimized Python client.
3. **Introduces negligible overhead:** Impacts median latency by only 20 microseconds.
4. **Allows for oblivious clients:** Can operate without the client having knowledge of its operation.

5.2 Current Limitations and Future Developments

Although functional and effective, the prototype developed has limitations that open the way for future research:

5.2.1 TCP Protocol Support

The current implementation only supports the UDP (User Datagram Protocol). Extending the system to TCP is a significant challenge, as it requires connection state management (sequence

numbers, congestion windows, retransmissions) to prevent the injection of duplicate packets from desynchronizing the flow or being interpreted as a TCP Injection attack. One possible solution involves integration with the kernel's conntrack module.

5.2.2 Adaptive Hedging

Currently, the timeout threshold () is fixed (e.g., 10 ms). In a real environment, network latency varies dynamically. A natural evolution of the system involves the use of eBPF maps to calculate real-time latency histograms, allowing the kernel to automatically adapt the hedging threshold (e.g., setting it to the current P95) without human intervention.

5.2.3 Integration with Kubernetes

To facilitate adoption in Cloud Native environments, the next step is to encapsulate the eBPF program in a CNI (Container Network Interface) plugin or in a Kubernetes DaemonSet, allowing hedging policies to be applied to specific Pods via simple YAML annotations, fully realizing the vision of a "Kernel Service Mesh."

Appendix A

Pseudocode of the Components

This appendix provides the algorithmic description of the three core components developed for this thesis: the Deterministic Chaos Server used for benchmarking, the eBPF programs running in the kernel, and the client-side implementations used to compare the hedging strategies.

A full working implementation can be found on my github.

A.1 Deterministic Chaos Server

The server simulates a microservice with tail latency issues. Crucially, the latency is injected deterministically based on the Request ID, ensuring that both Application and Kernel hedgers face exactly the same "bad" packets. It also implements a replay-memory to allow hedged requests (retries) to succeed immediately.

Algorithm 1 Deterministic Chaos UDP Server

```
1:
2:
3: procedure HandleRequest
4:
5:
       ‘‘‘                                                    ▷ Clean up old history
6:     History.RemoveIf(λt : now − t > TTL)
7:     delay ← 0.002                                          ▷ Default Fast Path: 2ms
8:     if (id (mod 100)) < 2 then                             ▷ 2% Fault Injection Rate
9:         if id ∈ History then
10:                                          ▷ Hedge detected! This ID was seen recently.
11:                                ▷ Skip stall to simulate successful retry on healthy node.
12:            delay ← 0.002
13:        else
14:                                                           ▷ First attempt: Inject Tail Latency
15:            delay ← 0.400                                  ▷ Stall: 400ms
16:            History.Insert(id, now)
17:        end if
18:    end if
19:    Sleep(delay)
20:    SendResponse(packet)
       ‘‘‘
21: end procedure
```

A.2 eBPF Programs (Kernel Space)

The core logic is split into two programs attached to the Traffic Control (TC) hook. The *Splitter* (Egress) clones packets, and the *Rescuer* (Ingress) re-injects them after the delay if necessary.

Algorithm 2 eBPF Splitter (TC Egress Hook)

```
1:
2: function HandleEgress
3:     if  or  then
4:         return
5:     end if
       ‘‘‘
6:     id ← LoadBytes(skb, PayloadOffset, 4)
7:     Scoreboard.Update(id, PENDING)
                                   ▷ Clone packet and redirect to Delay Line (veth)
8:     bpf_clone_redirect(skb, IFINDEX_VETH, 0)
9:     return TC_ACT_OK                              ▷ Let original packet pass
       ‘‘‘
10: end function
```

Algorithm 3 eBPF Rescuer (TC Ingress Hook)

```
1: function HandleIngressFromDelayLine
2:
3:
       ‘‘‘
4:     if status == COMPLETED then
5:         return TC_ACT_SHOT                    ▷ Response received, drop hedge
6:     end if
                                   ▷ Original request is still pending. Rescue it!
7:     EthHeader.Src ← LocalMac
8:     EthHeader.Dst ← GatewayMac
                                   ▷ Disable Checksum to prevent HW Offload drop
9:     UDP.Checksum ← 0
10:    bpf_redirect(IFINDEX_PHYSICAL, 0)
11:    return TC_ACT_REDIRECT
       ‘‘‘
12: end function
```

A.3 Client Implementations

This section contrasts the complexity of the Application-level hedging (User Space) versus the simplicity of the Kernel-level approach.

Algorithm 4 Client: Application-Hedged (Python/User Space)

```
1:  procedure SendRequestApp
2:
3:
        """                                                  ▷ User Space Timer Logic
4:      ready ← select.select([sock], [], [], 0.010)                    ▷ Wait 10ms
5:      if ready then
6:          return sock.recv()                                          ▷ Fast path success
7:      else
8:                                  ▷ Timeout! GC or Scheduler might have delayed this point.
9:          sock.send(packet(id))                                       ▷ Send Hedge
10:         ready ← select.select([sock], [], [], 1.0)
11:         if ready then
12:             return sock.recv()
13:         else
14:             raise TimeoutError
15:         end if
16:     end if
        """
17: end procedure
```

Algorithm 5 Client: Kernel-Hedged (eBPF Transparent)

```
1:  procedure SendRequestKernel
2:
        """                                  ▷ No hedging logic here. The kernel handles it.
3:      sock.send(packet(id))
4:      return sock.recv() TimeoutError
5:      raise TimeoutError
        """
6:  end procedure
```

References

[1] G. Fourny. *The Big Data Textbook - teaching large-scale databases in universities*. Aug. 2024. ISBN: 979-8337883137.

[2] J. Dean and L. A. Barroso. "The Tail at Scale". In: *Communications of the ACM* 56 (2013), pp. 74–80. URL: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[3] G. Linden. *Marissa Mayer at Web 2.0*. https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. Blog post. Nov. 2006.

[4] SentinelOne. *What is eBPF (Extended Berkeley Packet Filter)?* https://www.sentinelone.com/cybersecurity-101/cybersecurity/what-is-extended-berkeley-packet-filter-ebpf/. Accessed: 2026-02-14. 2023.