UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RAFAEL FREITAS CARDOSO

A Performance Comparison of Authentication and Authorization Patterns for Microservices Applications

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Jéferson Campos Nobre

Porto Alegre
August 2024

“You miss 100% of the shots you don’t take.”

— WAYNE GRETZKY

ACKNOWLEGDEMENTS

I would like to thank Prof. Jéferson Nobre, my thesis advisor, for all the guidance throughout the development of this work. Thanks to the Computer Networks Group at the Informatics Institute for providing me access and instructions on how to operate on their cluster. Thanks to my family, specially my parents Sílvio and Gisèlle, whose hard work allowed me to get here. Thanks to my girlfriend and soon to be fiancé Paola, for all the love and support. Last, but not least, I would like to thank my friends for all the good times we had during these years in college.

ABSTRACT

The microservices architecture has gained prominence in modern software development due to its flexibility, scalability, and resilience. However, ensuring robust security measures within microservices environments remains a challenge. This paper presents an empirical study evaluating authentication and authorization mechanisms in a microservices-based banking application. Three distinct versions of the baseline application were developed, each implementing a different authentication and authorization pattern: edge-level, centralized service-level, and decentralized service-level. Performance metrics including response time, resource consumption and network usage were collected and analyzed across 17 endpoints. Results indicate that decentralized authentication and authorization mechanisms generally outperform centralized approaches in terms of response time and resource efficiency. However, centralized solutions offer streamlined management and reduced storage overhead. The choice between centralized and decentralized architectures should consider factors such as storage scalability, data redundancy, and overall system performance. These findings contribute to the understanding of security implications in microservices architectures and provide insights for designing secure and efficient microservices-based systems.

Keywords: Microservices, Authentication, Authorization, Security, Performance.

RESUMO

A arquitetura de microsserviços tem ganhado destaque no desenvolvimento de software moderno devido à sua flexibilidade, escalabilidade e resiliência. No entanto, garantir medidas robustas de segurança dentro de ambientes de microsserviços continua sendo um desafio. Este trabalho apresenta um estudo empírico que avalia os mecanismos de autenticação e autorização em uma aplicação bancária baseada em microserviços. Foram desenvolvidas três versões distintas da aplicação básica, cada uma implementando um diferente padrão de autenticação e autorização para microsserviços: *edge-level*, *centralized service-level*, e *decentralized service-level*. Métricas de desempenho, incluindo tempo de resposta, consumo de recursos e uso de rede foram coletadas e analisadas ao longo de 17 endpoints. Os resultados indicam que os mecanismos de autenticação e autorização descentralizados geralmente superam as abordagens centralizadas em termos de tempo de resposta e eficiência de recursos. No entanto, as soluções centralizadas oferecem gerenciamento simplificado e redução da sobrecarga de armazenamento. A escolha entre arquiteturas centralizadas e descentralizadas deve considerar fatores como escalabilidade de armazenamento, redundância de dados e desempenho geral do sistema. Essas descobertas contribuem para a compreensão das implicações de segurança em arquiteturas de microserviços e fornecem insights para o projeto de sistemas baseados em microserviços seguros e eficientes.

Palavras-chave: Microsserviços, Autenticação, Autorização, Segurança, Desempenho.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CPU	Central Processing Unit
DB	Database
eBPF	extended Berkeley Packet Filter
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
MA	Microservices Architecture
MTLS	Mutual Transport Layer Security
PKI	Public Key Infrastructure
SD	Standard Deviation

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 Context

The microservices architecture (MA), characterized by the decomposition of applications into small, independent services, has emerged as a transformative paradigm in contemporary software development. This approach facilitates agility, scalability, and ease of deployment, allowing each service to evolve independently. The positive facets of microservices include accelerated development cycles and enhanced resilience to faults, as failures in one service do not necessarily cascade to others (FOWLER, 2014). However, this distributed nature, while offering numerous advantages, introduces complexities that necessitate careful consideration.

Security becomes a paramount concern as organizations increasingly embrace this approach. The very nature of a distributed architecture poses unique challenges, including the secure communication between services and protection against unauthorized access. Microservices, by their design, amplify the attack surface, demanding a comprehensive security strategy to safeguard the integrity of data and the privacy of users (DRAGONI et al., 2017). The motivation for this study lies in addressing these security challenges head-on, particularly focusing on the critical aspects of authentication and authorization.

A distributed environment allows different methods for validating the identity of entities seeking access and ascertaining their entitlements. Authentication, the process of verifying the legitimacy of users or services, serves as the gateway to secure interactions. In tandem, authorization determines the permissions granted to authenticated entities, ensuring they can only access resources and perform actions within their designated scope. Regardless of the chosen approach, a performance overhead is an inherent consideration. Security measures, aimed at fortifying the system against unauthorized access, inevitably introduce an additional layer of complexity.

Maintaining low overhead and achieving high performance are imperatives as we navigate through the security landscape of microservices. Given the dynamic nature of applications utilizing microservices, which often operate in resource-constrained environments (FERNANDO; WICKRAMAARACHCHI, 2022), optimizing the authentication and authorization processes becomes a compelling necessity. This paper addresses this imperative by presenting an experimental exploration of different microservice authentication and authorization patterns, aimed at discerning the most efficient approach.

1.2 Objectives

This study is designed with the overarching objective of assessing and contrasting the performance implications of distinct authentication and authorization patterns within the microservices architecture. The primary focus lies in empirically evaluating the efficiency of authentication and authorization strategies deployed in a simulated banking microservices application. Specifically, the study aims to measure and compare the performance across four versions of the application, each implementing varied approaches to authentication and authorization.

The first version, serving as a baseline, forgoes authentication and authorization altogether, providing a benchmark for performance evaluation. The second version introduces an edge-level approach, conducting authentication and authorization at the gateway before forwarding calls to services. Versions three and four distribute authentication and authorization to individual services, with version three employing a decentralized model where services independently handle these processes using their respective databases. Conversely, version four adopts a centralized approach, with services relying on an exclusive authentication and authorization service. The key objective is to experimentally determine the impact of these varied approaches on performance metrics, including response time, CPU utilization, memory consumption, and disk usage. Through rigorous analysis, this study aims to identify the authentication and authorization pattern that best balances security requirements with minimal performance overhead in the context of microservices architectures.

1.3 Motivation

The significance of this research delves into the critical examination of the strengths and weaknesses inherent in different authentication and authorization patterns. By scrutinizing the trade-offs between security and performance, this work aims to provide valuable insights for practitioners, developers, and architects navigating the intricate landscape of microservices security.

Furthermore, this study seeks to contribute to the broader discourse on microservices best practices. Identifying the pros and cons of each authentication and authorization pattern, in the light of their performance and resource consumption, holds inherent value. Such insights can guide the implementation of new security protocols, inform en-

hancements to existing methodologies, and empower decision-makers to make informed choices aligning with their specific use cases. In essence, the motivation for this research is rooted in advancing our comprehension of the security-performance equilibrium in microservices, offering actionable insights for the continual refinement of distributed systems.

1.4 Organization

This project is organized as follows: Chapter 2 provides a background, introducing concepts that are relevant to the understanding of the work. Chapter 3 summarizes some works related to this thesis. Chapter 4 describes the methodology used for building the baseline application, implementing its different versions, structuring a metric collection architecture and evaluating the performance of each one. Chapter 5 shows the results obtained from the experiments, and the performance comparisons between the patterns. At the end, Chapter 6 presents some conclusions drawn from the analysis of the results, as well as possible improvements and future work.

2 BACKGROUND

2.1 The Microservices Architecture

The microservices architecture represents a paradigm shift in software design and development, epitomizing the principle of architectural modularity. At its core, the MA advocates for the decomposition of complex applications into smaller, loosely coupled services, each encapsulating a specific functionality or business capability (NEWMAN, 2015). Unlike traditional monolithic architectures, where entire applications are built as a single, cohesive unit, the MA promotes the notion of independently deployable services, each with its own distinct purpose and responsibility.

The essence of microservices lies in its emphasis on autonomy, enabling development teams to work on individual services in isolation without being constrained by the dependencies inherent in monolithic architectures (FOWLER, 2014). This autonomy fosters rapid iteration, allowing teams to deploy updates and enhancements to specific services without affecting the functionality of the entire application. Additionally, the decentralized nature of microservices architecture enhances fault tolerance, as failures in one service are contained and do not propagate to other services.

Central to the philosophy of microservices is the concept of service boundaries, which delineate the scope and responsibility of each service within the broader application landscape (RICHARDSON, 2018). Services communicate with each other via lightweight protocols such as HTTP or messaging queues, facilitating seamless interaction while preserving modularity and independence.

The adoption of the MA is underpinned by a plethora of benefits, including enhanced scalability, improved fault isolation, and increased agility in software development (PAUTASSO et al., 2017). By breaking down applications into smaller, manageable components, organizations can adapt to changing requirements more effectively, iterate more rapidly, and deliver value to customers at a pace unparalleled by traditional monolithic approaches.

2.2 Authentication and Authorization Fundamentals

Authentication serves as the cornerstone of microservices security, verifying the identity of entities seeking access to resources or services within the system (ALMEIDA;

CANEDO, 2022). At its core, authentication validates the credentials provided by users or services, ensuring that they are who they claim to be. Authentication mechanisms commonly involve the exchange of credentials, such as usernames and passwords, tokens, or digital certificates, which are then verified against a trusted source, such as a directory service or identity provider.

In the context of microservices architectures, authentication mechanisms must accommodate the decentralized nature of the system, where services operate autonomously and interact with each other over distributed networks (RICHARDSON, 2018). This necessitates robust authentication protocols capable of securely validating identities across service boundaries while minimizing latency and overhead.

Authorization, on the other hand, governs the permissions granted to authenticated entities, determining what actions they are allowed to perform and which resources they can access within the system (ALMEIDA; CANEDO, 2022). Authorization mechanisms enforce access control policies based on predefined rules and roles, ensuring that only authorized entities can execute specific operations or access sensitive data.

The enforcement of authentication and authorization policies is critical for maintaining the security and integrity of microservices-based systems, particularly in environments where sensitive data and privileged operations are commonplace. However, achieving a balance between security and usability remains a challenge, as stringent authentication and authorization measures may introduce friction and complexity for end-users and developers alike.

2.3 Security Challenges in The Microservices Architecture

The MA introduces a myriad of security challenges stemming from its decentralized and distributed nature (MATEUS-COELHO; CRUZ-CUNHA; FERREIRA, 2021). Unlike monolithic architectures, where security concerns are often addressed holistically within a single application, microservices-based systems present unique complexities that require a nuanced approach to security.

One of the primary challenges in microservices security is the increased attack surface resulting from the proliferation of network endpoints and communication channels between services (DRAGONI et al., 2017). Each microservice exposes its own set of APIs, increasing the potential entry points for malicious actors to exploit vulnerabilities and launch attacks. Furthermore, the dynamic nature of microservices, with services

being deployed and scaled independently, adds another layer of complexity to security management and threat mitigation.

Securing communication between microservices is another critical concern in microservices architecture (PEREIRA-VALE et al., 2019). As services interact over distributed networks, ensuring the confidentiality, integrity, and authenticity of data in transit becomes paramount. Traditional security mechanisms such as transport layer encryption and message-level encryption are essential for safeguarding sensitive information from eavesdropping and tampering.

The MA also exacerbates the challenge of identity management and access control (HANNOUSSE; YAHIOUCHE, 2021). With numerous services operating autonomously, managing user identities and permissions across the system becomes increasingly complex. Granular access control policies must be implemented to ensure that only authorized entities can access sensitive resources and perform privileged operations.

Furthermore, the dynamic nature of microservices deployments introduces challenges in vulnerability management and patching. With services being continuously updated and scaled, maintaining a comprehensive inventory of software components and dependencies becomes demanding. This increases the risk of running outdated or vulnerable software, exposing the system to potential security threats.

2.4 Performance Considerations in Microservices

Performance optimization is a critical aspect of microservices architecture, as the distributed nature of the system introduces complexities that can impact overall system efficiency (LLOYD et al., 2018). Several factors influence the performance of microservices-based applications, ranging from communication overhead and network latency to resource utilization and scalability.

One of the primary considerations in microservices performance is the overhead associated with inter-service communication (DRAGONI et al., 2017). As services interact over distributed networks, each communication involves serialization, network transmission, and deserialization, all of which contribute to latency and overhead. Minimizing communication overhead through efficient message passing protocols and optimized network configurations is essential for maintaining responsive and scalable microservices-based systems.

Network latency also plays a significant role in microservices performance, partic-

ularly in environments where services may be deployed across geographically dispersed regions (HEINRICH et al., 2017). The physical distance between services can introduce latency in communication, impacting response times and overall system throughput. Strategies such as service placement optimization and content delivery network (CDN) integration can help mitigate the effects of network latency, ensuring optimal performance across distributed environments.

Resource utilization is another critical factor in microservices performance (HEINRICH et al., 2017). Each service consumes computational resources such as CPU, memory, and disk space, and efficient resource management is essential for maximizing system throughput and scalability. Containerization technologies such as Docker and orchestration platforms like Kubernetes enable fine-grained resource allocation and dynamic scaling, facilitating efficient resource utilization in microservices-based applications.

Scalability is a fundamental requirement for microservices architecture, allowing applications to handle increasing workloads and user demand without sacrificing performance (FOWLER, 2014). Horizontal scaling, achieved through the replication and distribution of services across multiple instances, is a common strategy for scaling microservices-based systems. However, achieving seamless scalability requires careful consideration of service dependencies, data consistency, and load balancing mechanisms.

2.5 Existing Authentication and Authorization Patterns

Authentication and authorization are fundamental aspects of microservices security, ensuring that only authenticated and authorized entities can access resources within the system (ALMEIDA; CANEDO, 2022). Various patterns and strategies have emerged to address these requirements, each offering unique benefits and trade-offs.

Centralized authentication and authorization mechanisms involve centralizing authentication and authorization logic within a dedicated service or identity provider (OWASP, 2017). This pattern offers centralized control and management of user identities and permissions, simplifying administration and enforcement of access control policies. However, centralized solutions may introduce single points of failure and scalability bottlenecks, particularly in large-scale distributed systems.

Decentralized authentication and authorization mechanisms distribute authentication and authorization logic across individual services, allowing each service to independently manage user identities and permissions (OWASP, 2017). This pattern offers greater

autonomy and flexibility, enabling services to adapt to changing requirements and scale independently. However, decentralized solutions may result in inconsistent enforcement of access control policies and increased complexity in managing distributed identities.

Edge authentication and authorization mechanisms offload authentication and authorization logic to the edge of the network, typically at the gateway or ingress controller (OWASP, 2017). This pattern offers low latency and efficient access control enforcement at the network perimeter, reducing the burden on individual services. Edge solutions are well-suited for scenarios where performance and scalability are paramount, such as high-throughput microservices-based applications. However, edge solutions may introduce dependencies on network infrastructure and limited flexibility in enforcing fine-grained access control policies.

Choosing the appropriate authentication and authorization pattern depends on various factors, including the specific requirements of the application, scalability considerations, and the desired level of autonomy and flexibility (PEREIRA-VALE et al., 2019). In practice, hybrid approaches that combine elements of centralized, decentralized, and edge authentication and authorization may offer the best balance of security and performance for microservices-based systems.

2.6 Technological Landscape of The Microservices Architecture

The MA is supported by a diverse ecosystem of technologies and platforms that enable developers to design, deploy, and manage distributed systems effectively (NEWMAN, 2015). Understanding the technological landscape surrounding microservices architecture is crucial for architects and developers seeking to leverage the benefits of decentralized systems.

Containerization technologies, such as Docker, have revolutionized the way distributed applications are packaged and deployed (SINGH; PEDDOJU, 2017). Containers provide lightweight, isolated environments that encapsulate individual services and their dependencies, ensuring consistency and portability across different environments. Containerization facilitates rapid development and deployment of microservices, streamlining the software delivery process.

Orchestration platforms, such as Kubernetes, have emerged as the de facto standard for managing containerized workloads in production environments (SAYFAN, 2019). Kubernetes provides robust features for automating deployment, scaling, and manage-

ment of microservices-based applications, enabling organizations to achieve high availability, resilience, and scalability. Kubernetes abstracts away the complexities of infrastructure management, allowing developers to focus on building and deploying applications without worrying about the underlying infrastructure.

Cloud services play a pivotal role in the development and operation of microservices architectures, offering scalable infrastructure and managed services for building and deploying applications (GUERRERO; LERA; JUIZ, 2018). Cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) offer a wide range of services tailored to the needs of microservices-based applications, including compute, storage, networking and database services. Leveraging cloud services allows organizations to scale their applications dynamically, optimize resource utilization, and reduce operational overhead.

In addition to containerization, orchestration, and cloud services, other technologies and frameworks, such as service mesh, API gateways, and observability tools, play crucial roles in microservices architectures (VIGGIATO et al., 2018). Service mesh technologies, like Istio and Linkerd, provide advanced features for managing service-to-service communication, traffic routing, and security policies. API gateways act as a central entry point for microservices-based applications, providing authentication, authorization, and request routing capabilities. Observability tools, such as Prometheus and Grafana, enable developers to monitor and troubleshoot microservices applications, gaining insights into performance, reliability, and resource utilization.

3 RELATED WORK

In this chapter, we review some of the important works and concepts that were developed around microservice security and performance in the recent years.

3.1 Microservice Security

Due to the evolving nature of this field, a multitude of recent studies have sought to elucidate and map the predominant practices pertaining to microservices security.

(NASAB et al., 2023) conducted an empirical study to identify and validate security practices for microservices systems. They manually analyzed 861 microservices security points from GitHub and Stack Overflow and ran a survey with 74 practitioners to evaluate the usefulness of the identified security practices.

With regard to authentication and authorization, the most consolidated mechanisms they discovered converge into two approaches for providing security: centralized and decentralized mechanisms. Centralized methods, such as using an identity microservice with *OpenID Connect*, establish a centralized point of control for authentication and authorization. They offer comprehensive management and verification of user identities while ensuring consistent security measures across the system.

On the other hand, decentralized mechanisms, such as using JWTs (*JSON Web Tokens*), distribute authentication responsibilities across microservices. This enables microservices to independently validate user identity claims, reducing the need for centralized coordination. These practices are opposed by nature and have associated advantages and disadvantages, making their use a design choice that depends on the application requirements.

(PEREIRA-VALE et al., 2019) performed a similar mapping study to identify the security mechanisms used in microservices-based systems described in the literature. The study selected 26 primary studies out of a total of 321 articles, and applied a rigorous protocol to extract, classify, and organize them. According to the study, the security aspects in microservices-based systems that are considered most important are **Authentication**, **Authorization** and **Credentials**, with their findings in regard to commonly used solutions also relying on centralized and decentralized mechanisms.

3.2 Mechanisms for Authentication and Authorization

Some recent studies have sought to identify, propose or implement security mechanisms related to authentication and authorization in the context of microservices.

(ALMEIDA; CANEDO, 2022) tried to identify in the literature the most common mechanisms used to deal with authentication and authorization challenges in a microservices architecture. It was found that there were few studies dealing with the subject, especially in practical order. The OAuth 2.0 protocol was the most mentioned (16 mentions), followed by JWT (14 mentions), API Gateway (14 mentions), Single Sign-On (8 mentions) and OpenID Connect (7 mentions). In terms of open-source solutions, they discovered it was possible to notice that the *Spring* ecosystem libraries were the most mentioned (*Spring Security*, *Spring Cloud*, *Spring Boot*, *Gateway Zuul* and *Eureka Server*).

(TRIARTONO; NEGARA; SUSSI, 2019) proposed the implementation of Role-Based Access Control on OAuth 2.0 as an authentication and authorization system. Specifically, the proposed model integrates OAuth 2.0 with RBAC, resulting in a more secure authentication and authorization system for microservices backend architecture. Their implementation was done using the *Laravel Framework* and the *Laravel Passport Library* with the author's proposed model. Since the model integrates OAuth 2.0, a centralized authorization framework, their solution can be categorized as a centralized security mechanism, which establishes a unified point of control, enabling administrators to enforce consistent security policies and access rules across different microservices.

(YARYGINA; BAGGE, 2018) proposed using *Mutual Transport Layer Security* (MTLS) with a self-hosted Public Key Infrastructure (PKI) as a method to protect internal service-to-service communication. Additionally, their work discusses the use of tokens and local authentication as trends in microservice security. These mechanisms align with common practices in authentication and authorization, such as using certificates for mutual authentication and issuing tokens for authorization purposes. The security mechanisms mentioned in the paper are primarily decentralized in nature. The authors argued that these approaches fit well with the second design principle of distributed systems: individual nodes should make decisions based on locally available information, facilitating loose coupling.

3.3 Microservice Performance Assessment

(SEDGHPOUR; TOWNEND, 2022) proposed using eBPF (*extended Berkeley Packet Filter*) as an efficient way to measure the performance of microservices. The eBPF is a technology that allows for dynamic tracing and monitoring of events within the Linux kernel. It provides a flexible way to analyze and manipulate network packets, system calls, and other events in real-time. They argued that this technology fits into measuring the performance of microservices by enabling deep visibility into system performance characteristics at the kernel level. This allows for the collection and monitoring of system metrics, providing insights into the behavior and performance of individual microservices.

(MIANO et al., 2021) conducted research on the use of eBPF-based network functions in microservices. They proposed a framework for leveraging eBPF to enhance the performance and flexibility of microservices architectures. In their work, they explored the capabilities of eBPF, such as packet filtering, event processing, and stateful processing using maps. They demonstrated how eBPF programs can be used to monitor network traffic, measure latency, and monitor resource utilization in microservices environments.

Both studies converge in the sense that it becomes possible to collect and analyze network performance metrics in real-time, enabling observability and the ability to make informed decisions regarding the implementation of microservices applications.

(COSTA et al., 2022) evaluated the performance impact of the **Retry** and **Circuit Breaker** patterns as implemented by two popular open-source resilience libraries: *Polly* for C# and *Resilience4j* for Java. The evaluation results showed that the Retry pattern could be more effective than the Circuit Breaker pattern in reducing contention for external resources, with both patterns causing a slight to moderate impact on execution time.

(SEDGHPOUR; KLEIN; TORDSSON, 2021) performed an empirical study to reveal the impact of different resiliency patterns, including circuit breaking and retry mechanisms, in various scenarios. They explored the performance implications of these patterns and proposed an adaptive controller to tune the circuit breaker in a service mesh in order to improve performance.

Despite operating within the realm of microservices, these studies have not specifically examined authentication and authorization patterns, focusing mostly on mechanisms related to fault tolerance.

4 METHODOLOGY

In this chapter, we delineate the methodology employed for evaluating the performance of microservices authentication and authorization patterns, as discussed in Chapter 2. We commence by elucidating the questions posed for this work. Subsequently, we introduce and expound upon the architecture of the developed application. Following this, we provide a comprehensive overview of the experiments conducted, detailing their setup and execution.

4.1 Research Questions

During the idealization of this work, that consists in comparing the **Edge-Level**, **Centralized Service-Level** and **Decentralized Service-Level** authentication and authorization patterns (as described in Section 2.5), some questions were posed. The main ones were:

- What is the overall impact of these patterns on application performance?
- What is the overall impact of these patterns on application resource consumption?
- How do the impacts of these patterns compare to each other? Which one(s) minimize overhead and resource consumption?

4.2 Host Application

The foundation of this study lies in the development of a microservices application tailored to simulate banking operations. Leveraging the Go programming language, the application draws strong inspiration from the open-source microservices application *Event Sourcing Examples* (RICHARDSON, 2017). While the original application was written in Java and utilized event-driven communication through a shared database, our adaptation in Go (CARDOSO, 2024) presents a distinct approach.

The choice of Go stems from its reputation as a high-performance language, well-suited for microservices applications. The event-driven approach gave place to employing HTTP requests for inter-service communication, as it aligns with prevalent practices in microservices architectures. This methodology seeks to encapsulate the essence of microservices applications while utilizing common technologies prevalent in the field.

The application encompasses core banking functionalities, including user and account management, transaction processing, balance tracking, and notification services. Comprising five distinct microservices — **Customer Service**, **Account Service**, **Transaction Service**, **Notification Service**, and **Balance Service** — the architecture is designed to foster seamless communication and independent service resolution. At the forefront of the application lies a gateway, orchestrating incoming requests and directing them to the appropriate services.

The application API architecture aggregates the individual service APIs, encompassing in total 17 distinct endpoints, each catering to specific functionalities across the microservices. Some endpoint calls are resolved independently by the services themselves, others require inter-service communication, involving up to four of the five services present in the architecture. Tables 4.1 and 4.2 show the endpoints provided by each service, and the endpoints depth (number of internal API calls performed), respectively.

Table 4.1: Service Endpoints

Service	**Endpoint**
Customer Service	createCustomer
Customer Service	deleteCustomer
Customer Service	getCustomer
Account Service	createAccount
Account Service	deleteAccount
Account Service	deleteAccountsByCustomer
Account Service	addToBalance
Account Service	subtractFromBalance
Account Service	getAccount
Account Service	getAccountsByCustomer
Transaction Service	transferAmount
Transaction Service	transferAmountAndNotify
Transaction Service	getTransaction
Notification Service	notify
Notification Service	getNotification
Balance Service	getBalanceByCustomer
Balance Service	getBalanceHistory

Source: The Author

The services are encapsulated within individual Docker containers, ensuring isolation and portability across diverse deployment environments. Moreover, each service maintains its dedicated PostgreSQL database instance, guaranteeing data integrity and encapsulated service logic. Operations within the services involve standard CRUD (Create, Read, Update, Delete) operations, executed on their respective databases.

Table 4.2: Endpoint Depths and Involved Services

Endpoint	Involved Services	Depth
createCustomer	Customer, Account	1
deleteCustomer	Account	0
getCustomer	Account	0
createAccount	Account	0
deleteAccount	Account	0
deleteAccountsByCustomer	Account	0
addToBalance	Account	0
subtractFromBalance	Account	0
getAccount	Account	0
getAccountsByCustomer	Account	0
transferAmount	Transaction, Account	2
transferAmountAndNotify	Transaction, Account, Notification, Customer	5
getTransaction	Transaction	0
notify	Notification, Account, Customer	2
getNotification	Notification	0
getBalanceByCustomer	Balance, Account	1
getBalanceHistory	Balance	0

Source: The Author

4.3 Employed Authentication and Authorization Mechanism

To enable the implementation of different versions of the application, a basic authentication and authorization method was devised. This method, while simple in design, lays the groundwork for enforcing access controls and validating user identities across the application.

A users table forms the cornerstone of the authentication and authorization mechanism. The table schema is presented in table 4.3. Each user entry in the database comprises a unique user ID, associated password, and boolean flags indicating the user's permissions for reading, writing, and deleting resources within the application.

For each API call within the application, the following authentication and authorization process is enforced:

- **Authentication:** The user ID and password provided in the API call parameters are validated against the corresponding entry in the users table. Successful authentication occurs when the provided credentials match an entry in the database.
- **Authorization:** Upon successful authentication, the user's permissions associated with the requested endpoint are evaluated. The application verifies whether the user possesses the requisite permissions to execute the requested operation. Permissions

include read, write, and delete capabilities, each governed by boolean flags in the user's database entry.

This simplistic yet effective authentication and authorization method ensures that only authenticated users with appropriate permissions can access and manipulate resources within the application. Table 4.4 shows the associated needed permissions with each API endpoint.

Table 4.3: Users Table Schema

Field Version	**Data Type**	**Constraints**
user_id	VARCHAR(255)	Primary Key
user_password	VARCHAR(255)	Not Null
can_read	BOOLEAN	Not Null
can_write	BOOLEAN	Not Null
can_delete	BOOLEAN	Not Null

Source: The Author

Table 4.4: Endpoint Needed Permissions

Endpoint	**Needed Permission**
createCustomer	WRITE
deleteCustomer	DELETE
getCustomer	READ
createAccount	WRITE
deleteAccount	DELETE
deleteAccountsByCustomer	DELETE
addToBalance	WRITE
subtractFromBalance	WRITE
getAccount	READ
getAccountsByCustomer	READ
transferAmount	WRITE
transferAmountAndNotify	WRITE
getTransaction	READ
notify	WRITE
getNotification	READ
getBalanceByCustomer	READ
getBalanceHistory	READ

Source: The Author

4.4 Application Versions

Four distinct versions of this application were developed to explore varying approaches to authentication and authorization within the microservices architecture.

- **Baseline Version:** This version represents the foundational iteration of the application and does not incorporate any authentication or authorization mechanisms. Requests are processed without any validation of user identity or permissions.
- **Edge Version:** In this iteration, authentication and authorization processes are centralized at the gateway level. Requests are authenticated and authorized before being forwarded to the respective services. This centralized approach aims to streamline security protocols and enforce uniform access controls across the application.
- **Centralized Version:** Centralizes authentication and authorization tasks through a dedicated authentication and authorization service. All services within this version communicate with the centralized service to validate user identities and enforce access controls. This centralized model fosters consistency in security enforcement while introducing a single point of authentication and authorization management.
- **Decentralized Version:** Authentication and authorization responsibilities are decentralized across the services in this version. Each service autonomously performs authentication and authorization tasks using its own database. This distribution of security tasks promotes service autonomy and reduces dependencies on external authentication services.

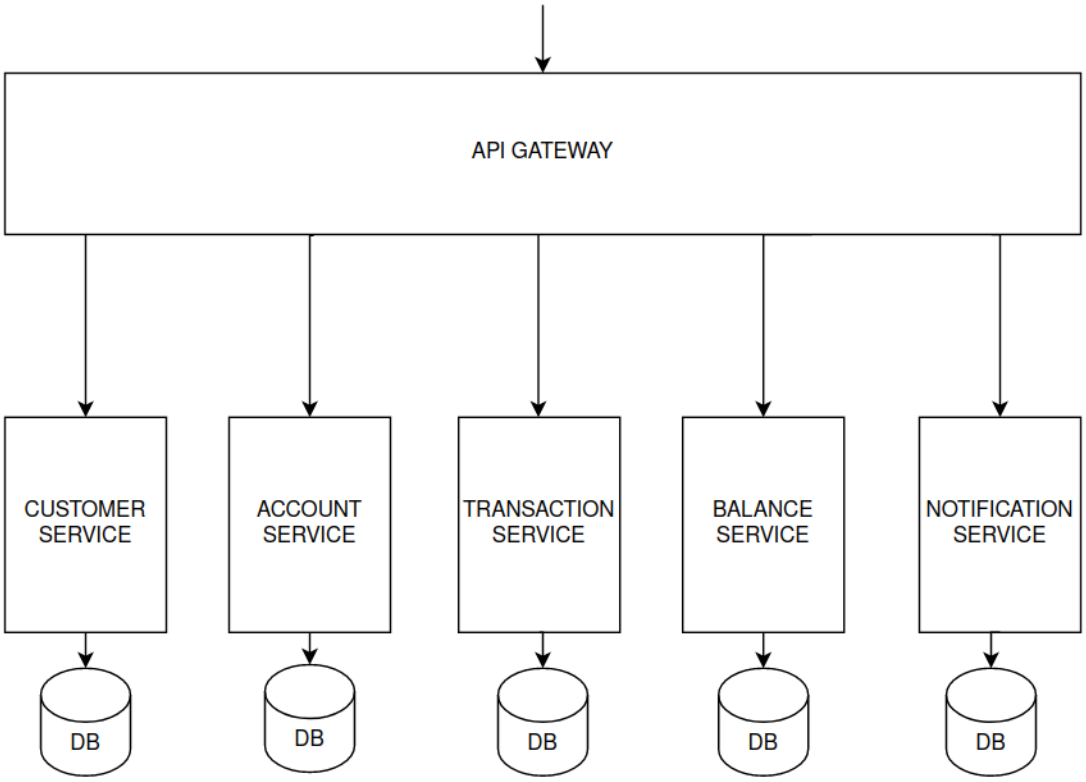
These distinct versions enable a comparative analysis of the authentication and authorization patterns, ranging from centralized to decentralized models, and their impact on application performance. Their respective architectures are displayed in figures 4.1 - 4.4.

4.5 Deploy Environment

All versions of the application underwent deployment, sequentially, within the cluster of the Computer Networks Group at the UFRGS Institute of Informatics, allocated within an exclusive namespace. Within this environment, each of the five services (along with the authentication service in the centralized version) operated as independent pods. Concurrently, dedicated pods were provisioned for the databases associated with each service. Additionally, a dedicated pod housed the gateway, serving as the primary entry point for incoming requests. All pods were uniformly provisioned with 1 CPU and 1 GB of memory, being also limited to these values.

To facilitate performance monitoring and resource utilization tracking with mini-

Figure 4.1: Baseline Application Architecture



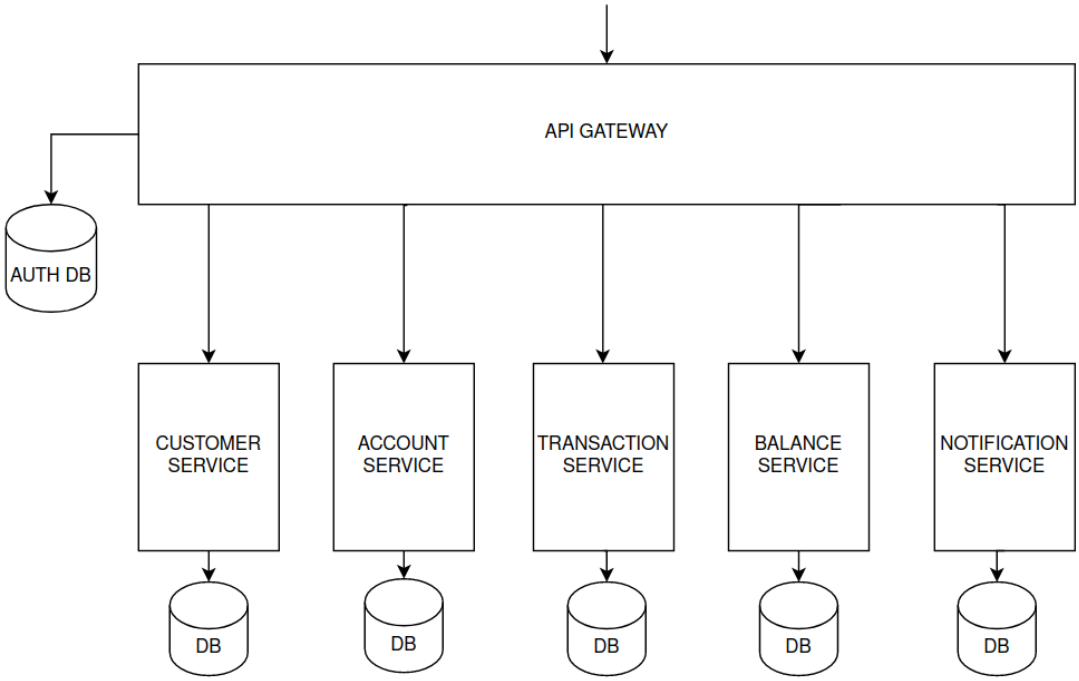
mal interference, the namespace was augmented with instances of Prometheus and Hubble. Prometheus, a widely adopted monitoring and alerting toolkit, served as the backbone for collecting and storing time-series data related to application metrics. Complementing Prometheus, Hubble operates on Cillium and eBPF frameworks, leveraging their capabilities for enhanced network observability. Combined, these tools offer crucial insights into performance, network and resource consumption metrics. Their integration proves especially apt in this experimental context, offering granular visibility into system behaviors without introducing significant overhead.

4.6 Performed Experiments

The evaluation experiments were conducted using a specialized Python tool developed for interfacing with the application gateway, following a black-box approach.

For each of the 17 endpoints within the application, a systematic barrage of 20,000 sequential requests was initiated to gauge system response under varying loads. The aim was to capture a spectrum of performance metrics, including minimum, maximum, and

Figure 4.2: Edge Version Application Architecture



average response times, alongside the request processing rate per minute. This process was iterated 10 times to ensure statistical robustness, resulting in 200,000 requests dispatched to each endpoint.

Data collection occurred at two levels: external and internal. Externally, the Python tool recorded temporal metrics to mirror user-level interactions with the system. Internally, the cluster's integrated monitoring tools facilitated granular insights into resource consumption and network usage patterns. This approach enabled a comprehensive evaluation of system performance from both user-facing and internal operational perspectives.

Efforts were made to minimize any potential overhead during data collection. Leveraging the monitoring tools integrated into the cluster, the resource and network metrics were obtained with minimal overhead and disruption to application operations.

This evaluation framework was systematically applied to each version of the application, enabling a detailed assessment of performance nuances across different authentication and authorization paradigms. Figure 4.5 illustrates the metric collection architecture implemented.

Figure 4.3: Centralized Version Application Architecture

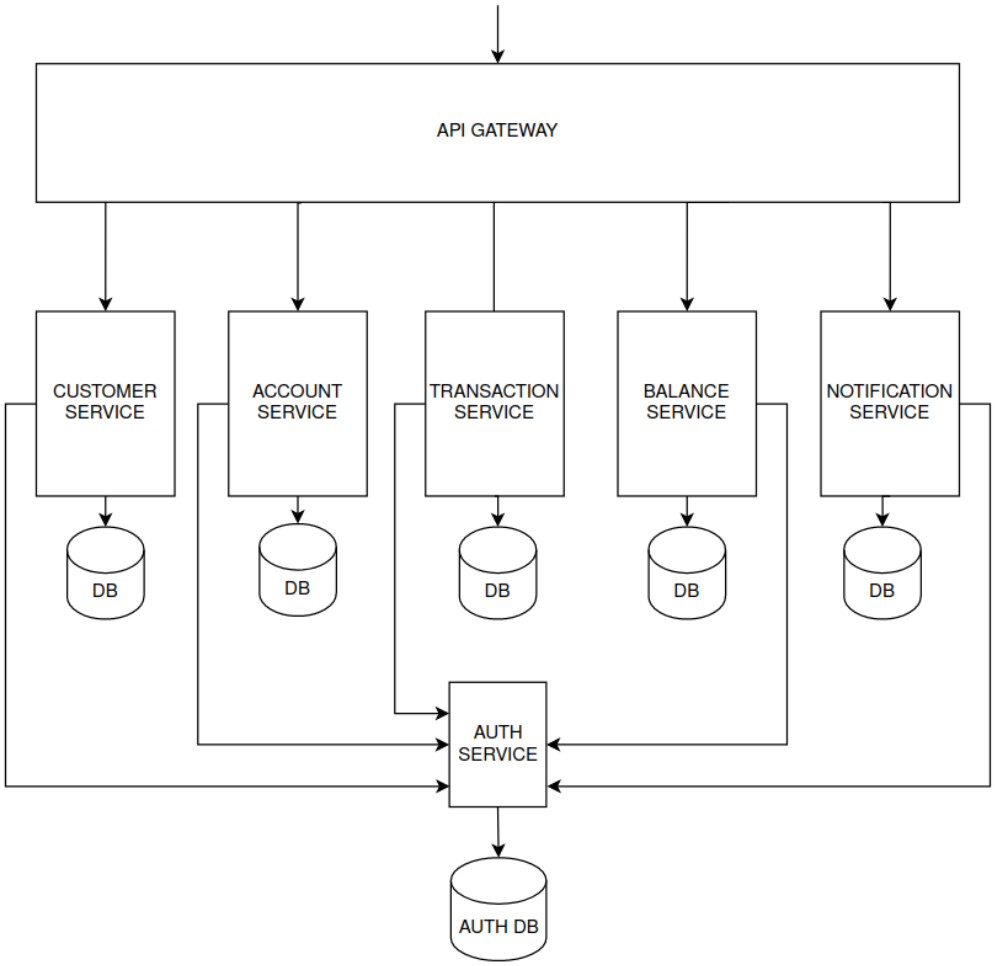


Figure 4.4: Decentralized Application Architecture

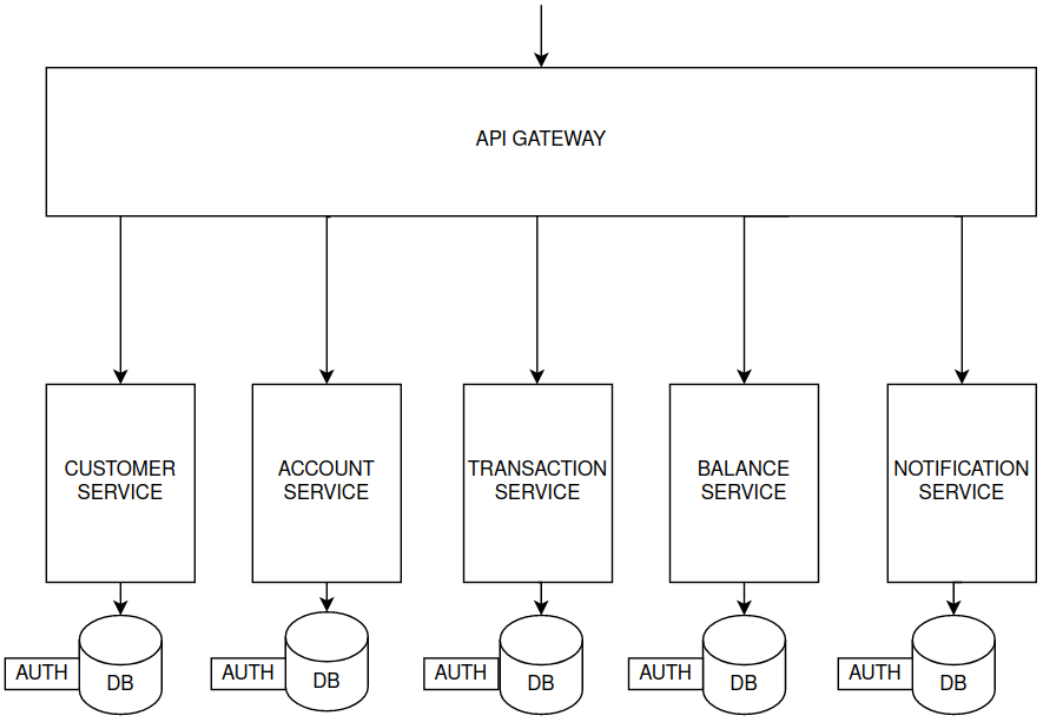
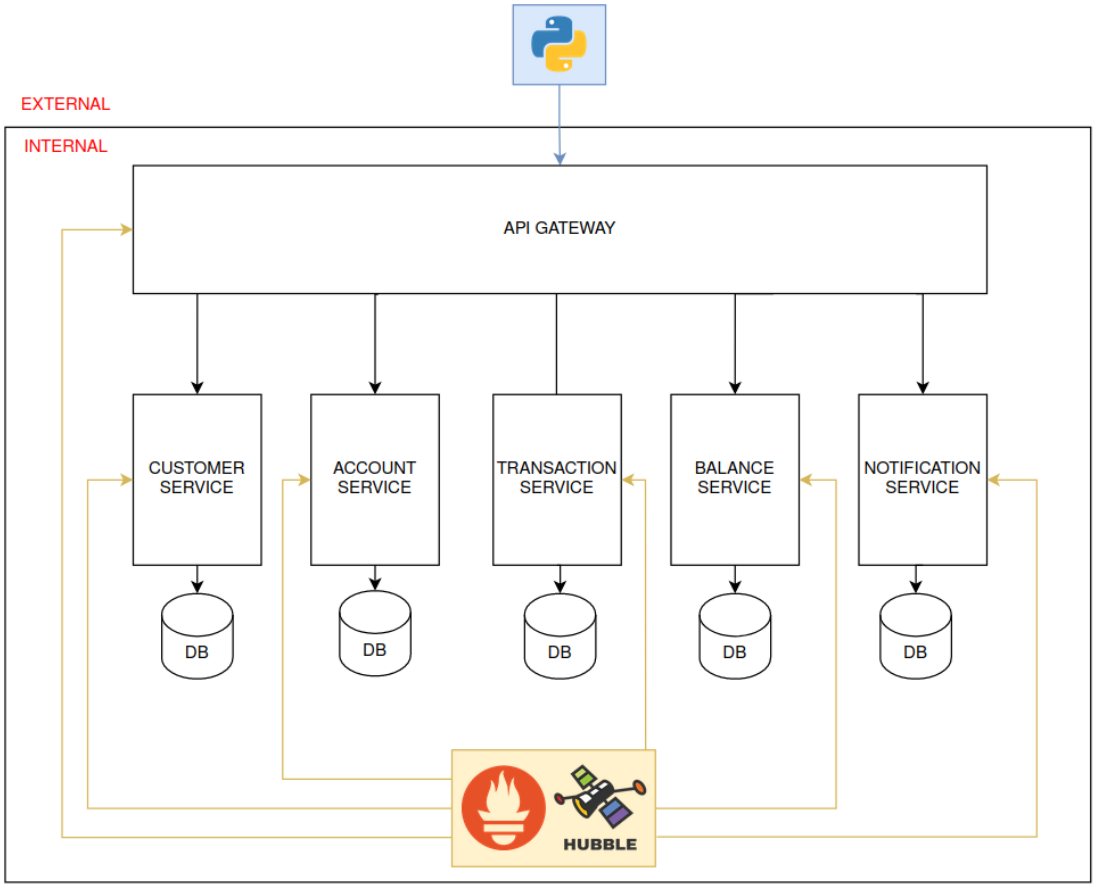


Figure 4.5: Metric Collection Architecture



5 RESULTS AND DISCUSSION

This chapter presents the results obtained from the executions of the experiments defined in Chapter 4. The results obtained for each application version are, then, used for comparison purposes.

In selecting endpoints for presenting the resulsts and performing analysis, we aimed for a balanced representation of the application's functionality while ensuring manageability in interpretation. Four endpoints were strategically chosen to span depths 0 (*createAccount*), 1 (*createCustomer*), 2 (*transferAmount*), and 5 (*transferAmountAndNotify*) within the microservices architecture, offering insights into varying levels of service interactions. By focusing on significant user interactions and transactional scenarios, these endpoints encapsulate essential functionalities, allowing for a holistic evaluation of system responsiveness and scalability.

These selections serve as exemplars of the application's behavior, offering valuable insights into operational dynamics. By concentrating on a subset of endpoints that emulate real-world interactions, we streamline the analysis process while maintaining the potential for extrapolating findings to the broader endpoint set. The chosen endpoints provide a focused lens through which to evaluate performance characteristics, facilitating a comprehensive understanding of the application's operational dynamics and scalability.

5.1 Performance

Tables 5.1 and 5.2 present mean performance metrics for the key endpoints across the four versions of the application: baseline, edge, centralized, and decentralized. Each metric is expressed both in terms of response time (in milliseconds) and requests per second. The metrics for the versions which perform authentication and authorization contain a percentage indicator in terms of the baseline version.

For the endpoint *createAccount*, we observe a consistent increase in response time across all versions compared to the baseline. The edge version shows a 24.41% increase, the centralized version exhibits a 37.55% increase, and the decentralized version demonstrates a 15.02% increase. Similarly, the requests per second metric decreases across all versions compared to the baseline, with the edge version performing at 80.45%, the centralized version at 73.13%, and the decentralized version at 87.24%.

Concerning the endpoint *createCustomer*, a similar trend is observed. The edge

Table 5.1: Mean Response Time

Endpoint - Application Version	**Response Time (ms)**	**SD**
createAccount - Baseline	4.26	0.0044
createAccount - Edge	5.30 (+24.41%)	0.0012
createAccount - Centralized	5.86 (+37.55%)	0.0012
createAccount - Decentralized	4.90 (+15.02%)	0.0012
createCustomer - Baseline	5.85	0.0012
createCustomer - Edge	6.98 (+19.31%)	0.0012
createCustomer - Centralized	8.45 (+44.44%)	0.0012
createCustomer - Decentralized	7.39 (+26.32%)	0.0012
transferAmount - Baseline	7.18	0.0012
transferAmount - Edge	8.58 (+19.49%)	0.0011
transferAmount - Centralized	10.74 (+49.58%)	0.0012
transferAmount - Decentralized	9.07 (+26.32%)	0.0012
transferAmountAndNotify - Baseline	10.39	0.0014
transferAmountAndNotify - Edge	11.97 (+15.20%)	0.0013
transferAmountAndNotify - Centralized	16.78 (+61.50%)	0.0023
transferAmountAndNotify - Decentralized	14.06 (+35.32%)	0.0014

Source: The Author

version experiences a 19.31% increase in response time and an 83.77% performance compared to the baseline. The centralized version demonstrates the highest increase in response time at 44.44% and the lowest performance at 69.33% of the baseline. Conversely, the decentralized version exhibits a 26.32% increase in response time and a 79.23% performance compared to the baseline.

For the endpoint *transferAmount*, the edge version shows a 19.49% increase in response time and an 83.66% performance compared to the baseline. The centralized version exhibits the highest increase in response time at 49.58% and the lowest performance at 66.87% of the baseline. In contrast, the decentralized version demonstrates a 26.32% increase in response time and a 79.19% performance compared to the baseline.

Lastly, the endpoint *transferAmountAndNotify* follows a similar pattern, with the edge version experiencing a 15.20% increase in response time and an 86.75% performance compared to the baseline. The centralized version shows the highest increase in response time at 61.50% and the lowest performance at 61.96% of the baseline. Conversely, the decentralized version exhibits a 35.32% increase in response time and a 73.93% performance compared to the baseline.

Overall, the centralized version consistently exhibits the highest total CPU usage and total memory consumption across all endpoints compared to the baseline. The edge version generally shows a moderate increase in resource consumption, while the de-

Table 5.2: Mean Requests Per Second

Endpoint - Application Version	**Requests Per Second**
createAccount - Baseline	228.78
createAccount - Edge	184.07 (80.45%)
createAccount - Centralized	167.31 (73.13%)
createAccount - Decentralized	199.61 (87.24%)
createCustomer - Baseline	169.48
createCustomer - Edge	141.98 (83.77%)
createCustomer - Centralized	117.51 (69.33%)
createCustomer - Decentralized	134.29 (79.23%)
transferAmount - Baseline	138.51
transferAmount - Edge	115.89 (83.66%)
transferAmount - Centralized	92.63 (66.87%)
transferAmount - Decentralized	109.69 (79.19%)
transferAmountAndNotify - Baseline	95.83
transferAmountAndNotify - Edge	83.14 (86.75%)
transferAmountAndNotify - Centralized	59.38 (61.96%)
transferAmountAndNotify - Decentralized	70.85 (73.93%)

Source: The Author

centralized version displays comparatively lower rises in resource usage. These results indicate that centralized authentication and authorization mechanisms impose heavier resource burdens on the system compared to both edge and decentralized approaches.

Furthermore, the centralized version's substantial increases in CPU and memory usage underscore potential scalability and efficiency challenges associated with centralized authentication and authorization architectures. In contrast, the edge version demonstrates a more moderate increase in resource consumption, suggesting a balanced approach between centralized and decentralized architectures. The decentralized version, on the other hand, displays the lowest rises in resource usage, indicating potential advantages in terms of resource efficiency and system scalability compared to both centralized and edge versions.

5.2 Resource Consumption

Tables 5.3 and 5.4 present mean resource consumption metrics for the key endpoints across the four versions of the application, aggregating each container metrics in order to provide a total amount. Resource consumption is measured in terms of total CPU usage (expressed as percentage of 1 dedicated CPU), and total memory usage (expressed in MiB). Following the stategy used earlier, metrics for the versions which perform au-

Table 5.3: Mean Total CPU Usage

Endpoint - Application Version	**Total CPU Usage (%)**	**SD**
createAccount - Baseline	46.54	0.38
createAccount - Edge	54.46 (+17.01%)	0.46
createAccount - Centralized	61.15 (+31.39%)	0.49
createAccount - Decentralized	53.51 (+14.97%)	0.44
createCustomer - Baseline	48.41	0.39
createCustomer - Edge	53.25 (+09.99%)	0.44
createCustomer - Centralized	63.66 (+31.50%)	0.50
createCustomer - Decentralized	55.31 (+14.25%)	0.45
transferAmount - Baseline	49.92	0.40
transferAmount - Edge	55.23 (+10.63%)	0.43
transferAmount - Centralized	63.79 (+27.78%)	0.49
transferAmount - Decentralized	56.01 (+26.32%)	0.46
transferAmountAndNotify - Baseline	52.83	0.41
transferAmountAndNotify - Edge	58.63 (+10.97%)	0.45
transferAmountAndNotify - Centralized	67.36 (+27.50%)	0.49
transferAmountAndNotify - Decentralized	59.17 (+12%)	0.45

Source: The Author

thentication and authorization contain a percentage indicator.

Examining the resource consumption for the *createAccount* endpoint, we observe a consistent increase in both CPU and memory usage across all versions compared to the baseline. The edge version exhibits a 17.01% increase in CPU usage and a 19.75% increase in memory usage. Similarly, the centralized and decentralized versions show comparable trends, with both experiencing higher resource consumption levels compared to the baseline.

For the *createCustomer* endpoint, a similar pattern emerges. The edge version demonstrates a 9.99% increase in CPU usage and an 18.89% increase in memory usage compared to the baseline. Notably, the centralized version exhibits the highest increase in resource consumption, with a 31.50% rise in CPU usage and a 24.81% increase in memory usage. The decentralized version also shows elevated resource consumption levels compared to the baseline.

Regarding the *transferAmount* endpoint, all versions display increased resource consumption metrics compared to the baseline. The edge version shows a 10.63% increase in CPU usage and a 19.17% increase in memory usage. Similarly, both the centralized and decentralized versions exhibit higher resource consumption levels, with the centralized version demonstrating the most significant increase.

Analyzing the *transferAmountAndNotify* endpoint, we observe a consistent trend

Table 5.4: Mean Total Memory Usage

Endpoint - Application Version	**Total Memory Usage (MiB)**	**SD**
createAccount - Baseline	171.74	0.12
createAccount - Edge	205.67 (+19.75%)	0.17
createAccount - Centralized	214.82 (+25.08%)	0.18
createAccount - Decentralized	210.41 (+22.51%)	0.18
createCustomer - Baseline	173.84	0.13
createCustomer - Edge	206.68 (+18.89%)	0.17
createCustomer - Centralized	216.98 (+24.81%)	0.18
createCustomer - Decentralized	211.29 (+21.54%)	0.18
transferAmount - Baseline	175.69	0.12
transferAmount - Edge	209.37 (+19.17%)	0.17
transferAmount - Centralized	218.49 (+24.36%)	0.18
transferAmount - Decentralized	213.68 (+21.62%)	0.18
transferAmountAndNotify - Baseline	177.41	0.12
transferAmountAndNotify - Edge	210.96 (+18.91%)	0.17
transferAmountAndNotify - Centralized	219.99 (+24%)	0.19
transferAmountAndNotify - Decentralized	215.21 (+21.30%)	0.18

Source: The Author

across versions, with higher resource consumption compared to the baseline. The edge version demonstrates a 10.97% increase in CPU usage and an 18.91% increase in memory usage. The centralized version shows the highest increase in both CPU and memory usage among all versions, further emphasizing the impact of centralized authentication and authorization mechanisms on resource consumption.

The resource consumption analysis reveals notable trends across the different versions of the application. Firstly, the centralized version consistently demonstrates elevated total CPU usage and memory consumption compared to the baseline configuration. This indicates that centralized authentication and authorization mechanisms place heavier demands on system resources than both edge and decentralized approaches.

Moreover, the substantial increases in CPU and memory usage observed in the centralized version underscore potential scalability and efficiency challenges inherent to centralized authentication and authorization architectures. In contrast, the edge version exhibits a more moderate rise in resource consumption, suggesting a balanced approach between centralized and decentralized architectures in terms of resource utilization.

Conversely, the decentralized version presents the lowest increases in resource usage, implying potential advantages in resource efficiency and system scalability compared to both centralized and edge versions.

5.3 Network Usage

Table 5.5: Mean Network Usage

Endpoint - Application Version	Total Transmitted Data (MB/s)	SD
createAccount - Baseline	1.12	0.01
createAccount - Edge	1.26 (+12.5%)	0.01
createAccount - Centralized	1.46 (+30.35%)	0.10
createAccount - Decentralized	1.37 (+22.32%)	0.05
createCustomer - Baseline	1.26	0.02
createCustomer - Edge	1.44 (+14.28%)	0.03
createCustomer - Centralized	1.58 (+25.39%)	0.09
createCustomer - Decentralized	1.55 (+23.01%)	0.08
transferAmount - Baseline	1.30	0.02
transferAmount - Edge	1.52 (+16.92%)	0.05
transferAmount - Centralized	1.69 (+30%)	0.12
transferAmount - Decentralized	1.62 (+24.61%)	0.10
transferAmountAndNotify - Baseline	1.44	0.03
transferAmountAndNotify - Edge	1.64 (+13.88%)	0.06
transferAmountAndNotify - Centralized	1.92 (+33.33%)	0.15
transferAmountAndNotify - Decentralized	1.79 (+24.30%)	0.11

Source: The Author

Table 5.5 provides insights into the mean network usage across the key endpoints across the four versions of the application. Network usage is measured in terms of total transmitted data in megabytes per second (MB/s). Once again, there is a percentage indicator in terms of the baseline version.

For the endpoint *createAccount*, we observe consistent increases in total transmitted data across all versions compared to the baseline. The edge version exhibits a 12.5% increase, the centralized version shows a 30.35% increase, and the decentralized version demonstrates a 22.32% increase.

Similarly, for the endpoint *createCustomer*, there is a consistent upward trend in total transmitted data across all versions compared to the baseline. The edge version displays a 14.28% increase, the centralized version shows a 25.39% increase, and the decentralized version demonstrates a 23.01% increase.

Concerning the endpoint *transferAmount*, a similar pattern emerges, with all versions showing higher total transmitted data compared to the baseline. The edge version exhibits a 16.92% increase, the centralized version shows a 30% increase, and the decentralized version demonstrates a 24.61% increase.

Lastly, for the endpoint *transferAmountAndNotify*, increases in total transmitted

data are observed across all versions compared to the baseline. The edge version displays a 13.88% increase, the centralized version shows a 33.33% increase, and the decentralized version demonstrates a 24.30% increase.

The network usage analysis unveils significant patterns among the different application versions. Primarily, the centralized version consistently records the highest total transmitted data across all endpoints compared to the baseline configuration. This suggests that centralized authentication and authorization mechanisms contribute to elevated network usage compared to both edge and decentralized approaches.

Moreover, the substantial increases in total transmitted data observed in the centralized version underscore potential scalability and efficiency challenges inherent to centralized authentication and authorization architectures. Conversely, the edge version demonstrates a more moderate rise in network usage, indicative of a balanced approach between centralized and decentralized architectures in terms of network efficiency.

In contrast, the decentralized version exhibits the lowest rises in transmitted data, implying potential advantages in network efficiency and system scalability compared to both centralized and edge versions.

5.4 Disk Usage

Table 5.6: Disk Usage

Application Version	**Total Disk Usage (MB)**
Baseline	242.9
Edge	322 (+32.56%)
Centralized	322 (+32.56%)
Decentralized	394 (+62.20%)

Source: The Author

Table 5.6 illustrates the total disk space utilized by different versions of the application, reflecting the storage requirements of user data across the experiment. In comparison to the baseline, all versions demonstrate an increase in disk usage. Both the edge and centralized versions exhibit identical disk usage levels, amounting to 132.56% of the baseline. Conversely, the decentralized version displays the highest disk usage, accounting for 162.20% of the baseline.

The observed disparity in disk usage among the versions can be attributed to the underlying architecture of the authentication and authorization mechanisms. In the edge

and centralized versions, user data is centralized within a single database instance, resulting in a more efficient storage utilization model. However, the decentralized version necessitates the replication of user data across multiple service databases, contributing to higher disk space consumption.

6 CONCLUSIONS

The conclusions drawn from this comprehensive study shed light on the intricate dynamics of authentication and authorization mechanisms within microservices architectures. Through meticulous experimentation and analysis, several key insights have surfaced, shaping the way forward for practitioners and researchers alike.

First and foremost, the performance evaluation of the microservices application across different authentication and authorization patterns has unearthed nuanced trends. The collected data showed that the choice of authentication and authorization mechanisms significantly influences both the performance and resource consumption profiles of microservices applications. The centralized authentication and authorization approach consistently exhibited the highest resource utilization across all endpoints, indicating potential scalability and efficiency challenges. Conversely, the decentralized approach emerged as a promising alternative, displaying lower resource consumption and better performance outcomes compared to both centralized and edge configurations.

Furthermore, the network usage analysis highlighted the impact of authentication and authorization mechanisms on network efficiency. The centralized approach led to heightened network usage, posing scalability concerns and underscoring the need for optimization strategies in centralized architectures. In contrast, the decentralized approach demonstrated superior network efficiency and system scalability, reaffirming its viability in microservices environments.

Moreover, the disk usage metrics obtained highlight the diverse storage demands associated with different authentication and authorization patterns. The centralized and edge versions, which maintain a single copy of user data in the authentication database, exhibit comparable levels of disk usage. In contrast, the decentralized version, which necessitates multiple copies of user data distributed across service databases, incurs substantially higher disk usage.

These findings underscore the importance of carefully considering the trade-offs associated with different authentication and authorization strategies in microservices architectures. While centralized mechanisms may offer administrative convenience and a diminished need for storage space, they come at the cost of increased resource utilization and reduced system scalability. On the other hand, decentralized approaches offer better performance and network efficiency, albeit with added complexity in implementation and management.

In addition to the performance and resource consumption metrics, it's imperative to acknowledge the security implications inherent in the choice of authentication and authorization mechanisms. While the edge version of the application demonstrated superior performance and resource efficiency compared to other patterns, it is crucial to recognize its inherent security trade-offs. Unlike the centralized and decentralized approaches, which offer service-level authentication and authorization, the edge version operates only at the edge level. This enables vulnerabilities in terms of security, particularly in zero-trust environments, as it does not contemplate defense-in-depth. In a zero-trust architecture, where every user and device is treated as untrusted, the granularity and robustness of authentication mechanisms play a pivotal role in safeguarding against potential threats and unauthorized access attempts.

Therefore, while the edge-level pattern may offer compelling performance advantages, organizations must carefully weigh the trade-offs between performance and security when selecting authentication and authorization mechanisms for microservices architectures. In security-critical environments or those adhering to zero-trust principles, the benefits of service-level authentication and authorization provided by centralized and decentralized approaches outweigh the performance gains offered by the edge paradigm. As such, the choice of authentication and authorization mechanisms should be aligned with the organization's security posture, regulatory requirements, and risk tolerance levels.

In extending the scope of this study, future investigations could delve into various parameters affecting the performance and security of the patterns. Exploring diverse authentication mechanisms, including token-based or multi-factor authentication, may shed light on their impact on system responsiveness and resource utilization. Similarly, analyzing the influence of factors such as password length and encryption integration could offer valuable insights into enhancing system security without compromising performance.

Stress testing could also be employed to assess whether variations in load, database usage and traffic conditions can alter the trends observed in the comparison of the authentication and authorization patterns. Additionally, examining scenarios involving service and database replication can provide relevant perspectives into the patterns performance under varied circumstances.

The assessment of dynamic user data management and synchronization mechanisms between databases is another possible development. Investigating synchronization protocols and their impact on system performance can guide the development of robust distributed architectures.

Lastly, evaluating optimizations within the studied patterns, such as implementing caching mechanisms in the authentication service and enabling data sharing between services to mitigate the replication of user credentials, presents another avenue for exploration. Analyzing the effectiveness of these optimizations in improving system responsiveness and reducing resource overhead can inform future architectural decisions and optimizations, ensuring that performance remains a focal point in the evolution of microservices-based applications.

REFERENCES

ALMEIDA, M. G. de; CANEDO, E. D. Authentication and authorization in microservices architecture: A systematic literature review. **Applied Sciences**, MDPI, v. 12, n. 6, p. 3023, 2022.

CARDOSO, R. **Microservices Auth Benchmark**. 2024. Available from Internet: <https://github.com/rfcardoso07/microservices-auth-benchmark>.

COSTA, T. et al. Avaliação de desempenho de dois padrões de resiliência para microsserviços: Retry e circuit breaker. In: **Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2022. p. 517–530. ISSN 2177-9384. Available from Internet: <https://sol.sbc.org.br/index.php/sbrc/article/view/21194>.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. **Present and ulterior software engineering**, Springer, p. 195–216, 2017.

FERNANDO, R.; WICKRAMAARACHCHI, D. Performance optimization of microservice applications under resource constrained environments. In: **2022 International Research Conference on Smart Computing and Systems Engineering (SCSE)**. [S.l.: s.n.], 2022. v. 5, p. 309–313.

FOWLER, M. **Microservices: a definition of this new architectural term**. 2014. Available from Internet: <https://martinfowler.com/articles/microservices.html>.

GUERRERO, C.; LERA, I.; JUIZ, C. Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. **The Journal of Supercomputing**, Springer, v. 74, n. 7, p. 2956–2983, 2018.

HANNOUSSE, A.; YAHIOUCHE, S. Securing microservices and microservice architectures: A systematic mapping study. **Computer Science Review**, Elsevier, v. 41, p. 100415, 2021.

HEINRICH, R. et al. Performance engineering for microservices: research challenges and directions. In: **Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion**. [S.l.: s.n.], 2017. p. 223–226.

LLOYD, W. et al. Serverless computing: An investigation of factors influencing microservice performance. In: IEEE. **2018 IEEE international conference on cloud engineering (IC2E)**. [S.l.], 2018. p. 159–169.

MATEUS-COELHO, N.; CRUZ-CUNHA, M.; FERREIRA, L. G. Security in microservices architectures. **Procedia Computer Science**, Elsevier, v. 181, p. 1225–1236, 2021.

MIANO, S. et al. A framework for ebpf-based network functions in an era of microservices. **IEEE Transactions on Network and Service Management**, v. 18, n. 1, p. 133–151, 2021.

NASAB, A. R. et al. An empirical study of security practices for microservices systems. **Journal of Systems and Software**, Elsevier, v. 198, p. 111563, 2023.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. [S.l.]: "O'Reilly Media, Inc.", 2015.

OWASP. **Microservices Security Cheat Sheet**. 2017. Available from Internet: <https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html>.

PAUTASSO, C. et al. Microservices in practice, part 1: Reality check and service design. **IEEE Software**, v. 34, n. 1, p. 91–98, 2017.

PEREIRA-VALE, A. et al. Security mechanisms used in microservices-based systems: A systematic mapping. In: **2019 XLV Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2019. p. 01–10.

RICHARDSON, C. **Event Sourcing Examples**. 2017. Available from Internet: <https://github.com/cer/event-sourcing-examples>.

RICHARDSON, C. **Microservices patterns: with examples in Java**. [S.l.]: Simon and Schuster, 2018.

SAYFAN, G. **Hands-On Microservices with Kubernetes: Build, deploy, and manage scalable microservices on Kubernetes**. [S.l.]: Packt Publishing Ltd, 2019.

SEDGHPOUR, M. R. S.; KLEIN, C.; TORDSSON, J. Service mesh circuit breaker: From panic button to performance management tool. In: **Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems**. New York, NY, USA: Association for Computing Machinery, 2021. (HAOC '21), p. 4–10. ISBN 9781450383363. Available from Internet: <https://doi.org/10.1145/3447851.3458740>.

SEDGHPOUR, M. R. S.; TOWNEND, P. Service mesh and ebpf-powered microservices: A survey and future directions. In: **2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. [S.l.: s.n.], 2022. p. 176–184.

SINGH, V.; PEDDOJU, S. K. Container-based microservice architecture for cloud applications. In: **2017 International Conference on Computing, Communication and Automation (ICCCA)**. [S.l.: s.n.], 2017. p. 847–852.

TRIARTONO, Z.; NEGARA, R. M.; SUSSI. Implementation of role-based access control on oauth 2.0 as authentication and authorization system. In: **2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)**. [S.l.: s.n.], 2019. p. 259–263.

VIGGIATO, M. et al. Microservices in practice: A survey study. **arXiv preprint arXiv:1808.04836**, 2018.

YARYGINA, T.; BAGGE, A. H. Overcoming security challenges in microservice architectures. In: IEEE. **2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)**. [S.l.], 2018. p. 11–20.

APPENDIX A — RESULT PLOTS

Figure A.1: Mean Response Time - *createAccount*

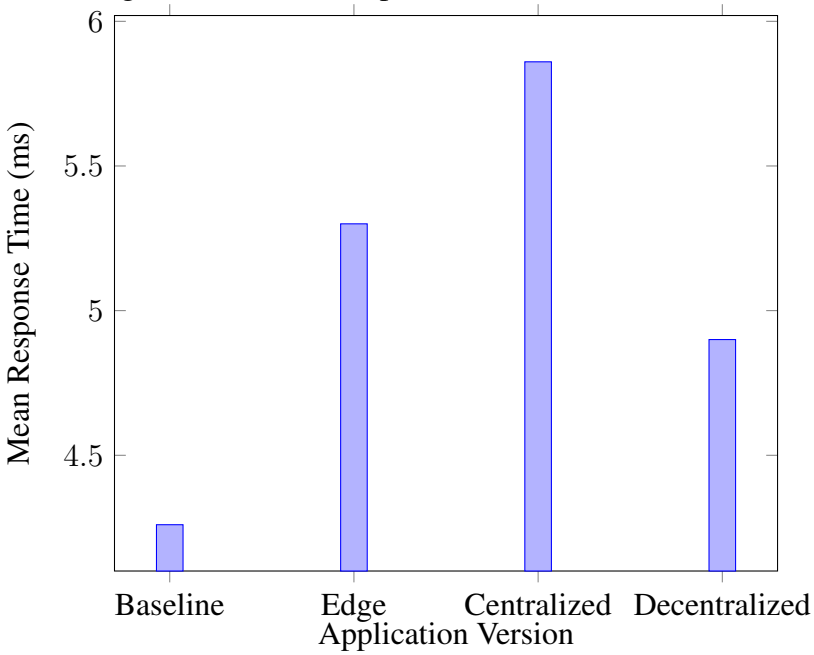


Figure A.2: Mean Response Time - *createCustomer*

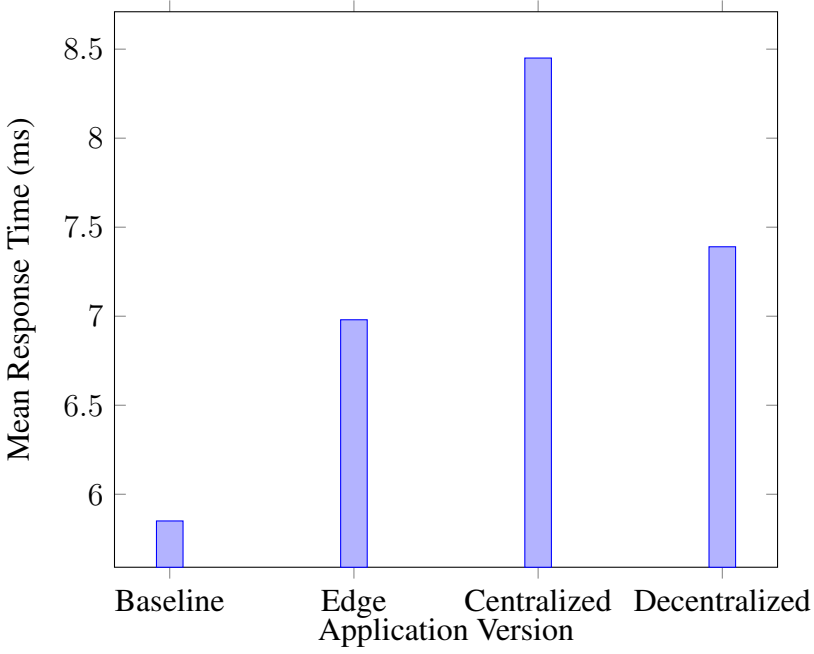


Figure A.3: Mean Response Time - *transferAmount*

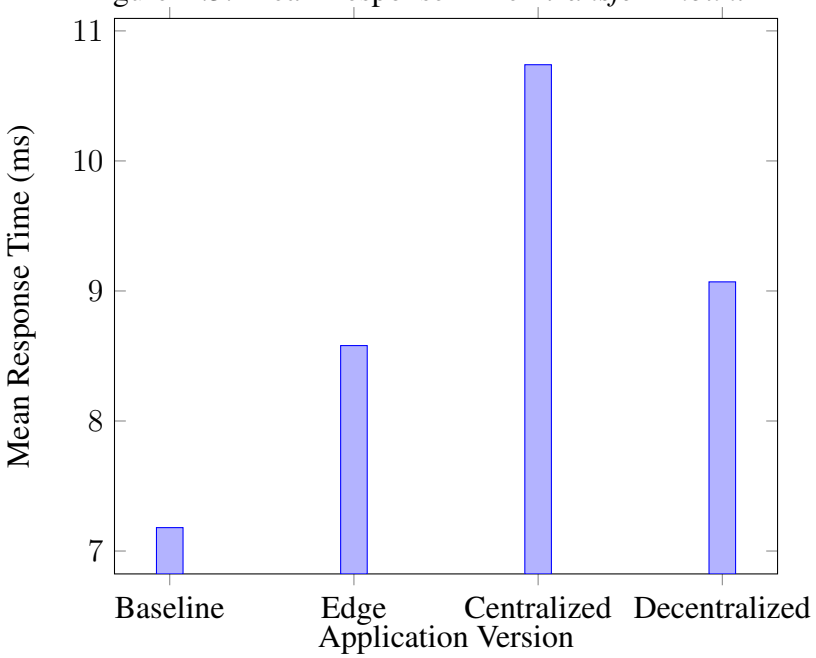


Figure A.4: Mean Response Time - *transferAmountAndNotify*

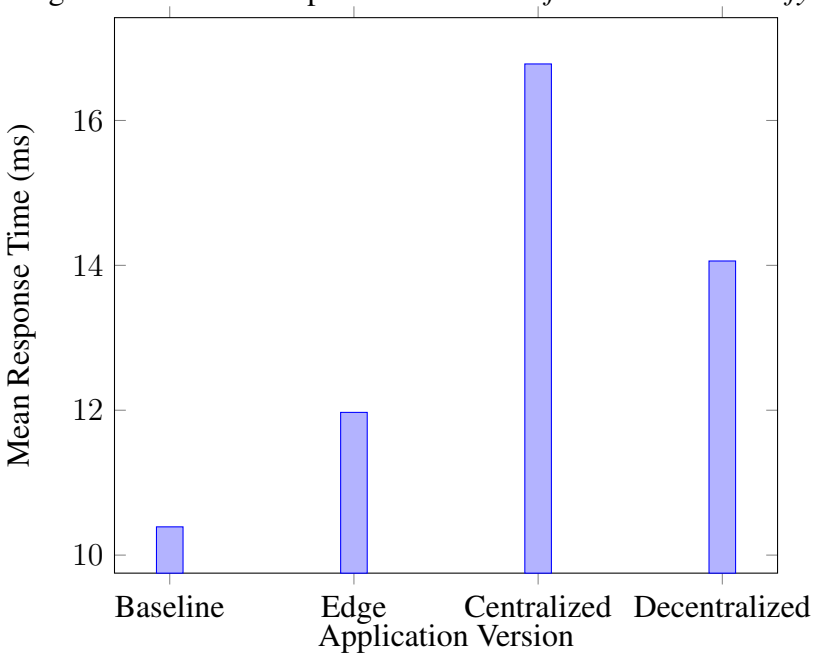


Figure A.5: Mean Total CPU Usage - *createAccount*

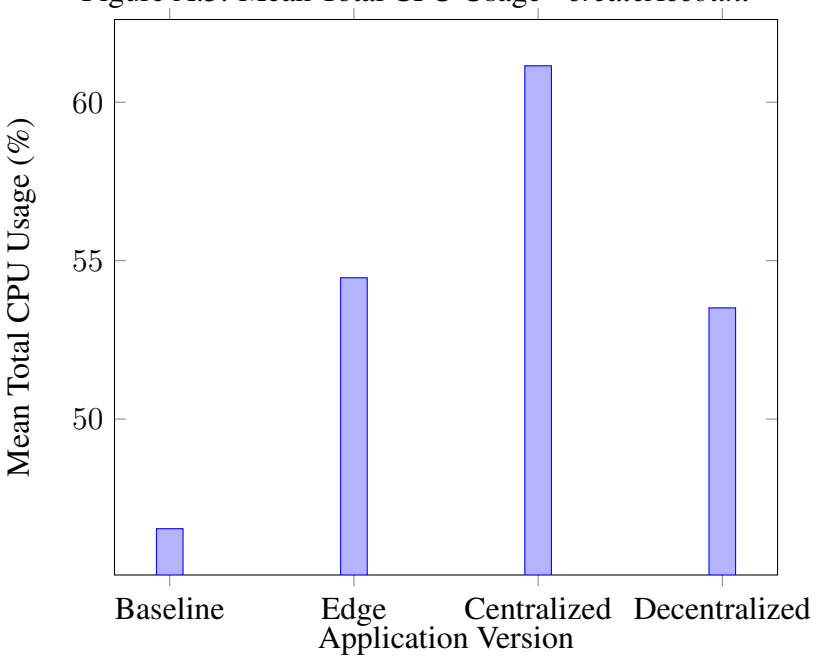


Figure A.6: Mean Total CPU Usage - *createCustomer*

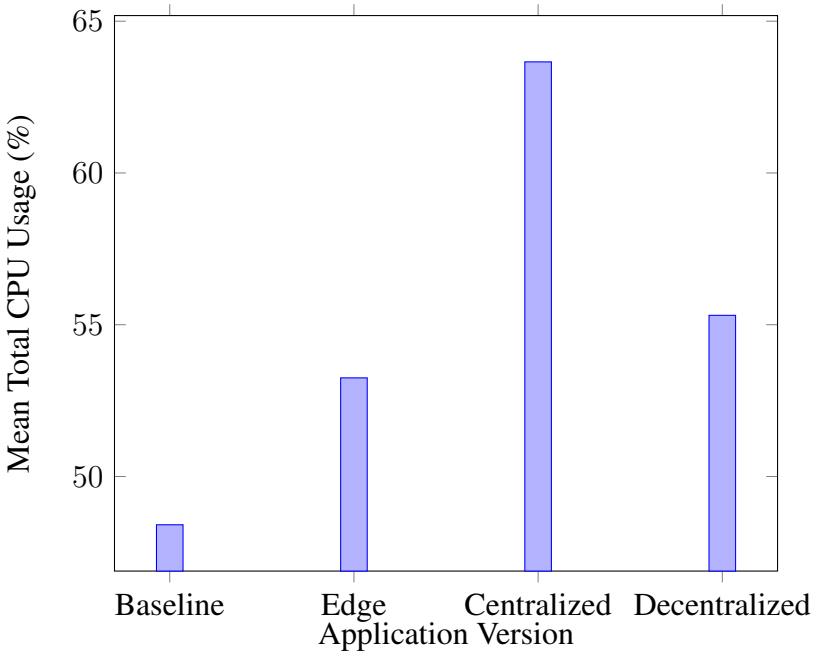


Figure A.7: Mean Total CPU Usage - *transferAmount*

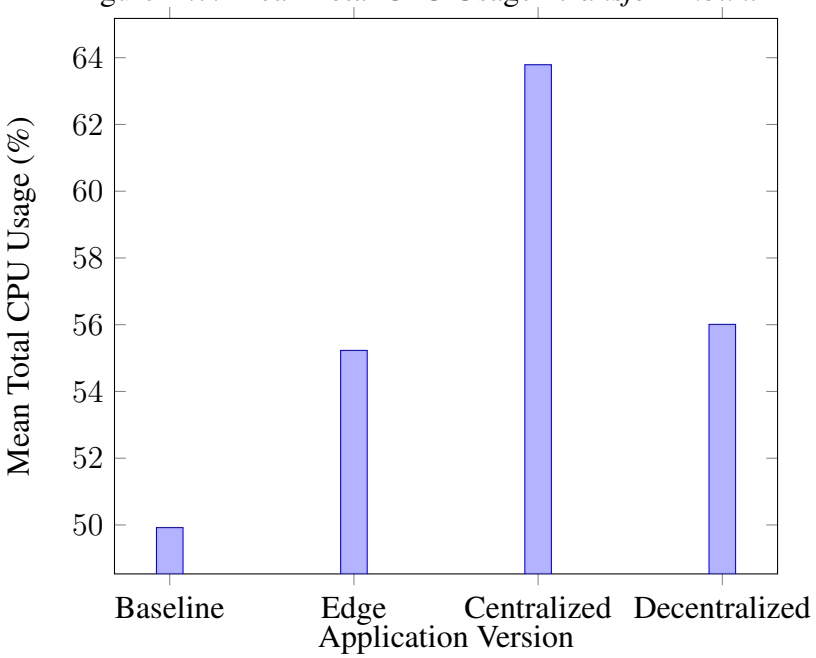


Figure A.8: Mean Total CPU Usage - *transferAmountAndNotify*

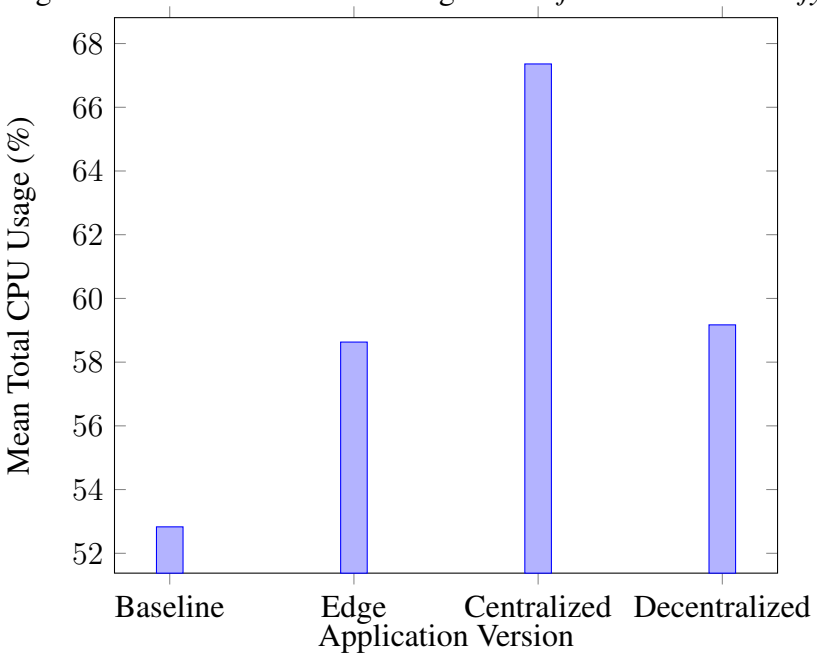


Figure A.9: Mean Total Memory Usage - *createAccount*

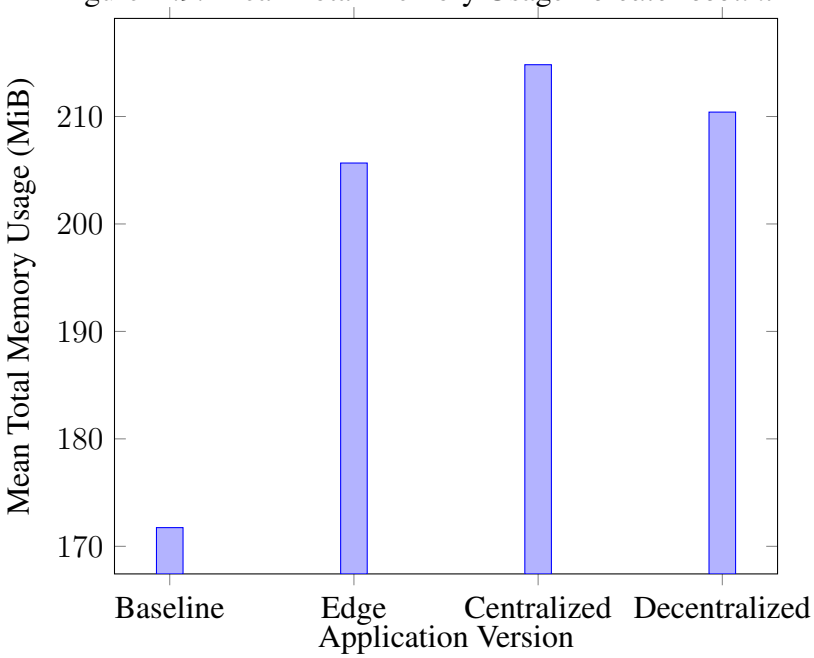


Figure A.10: Mean Total Memory Usage - *createCustomer*

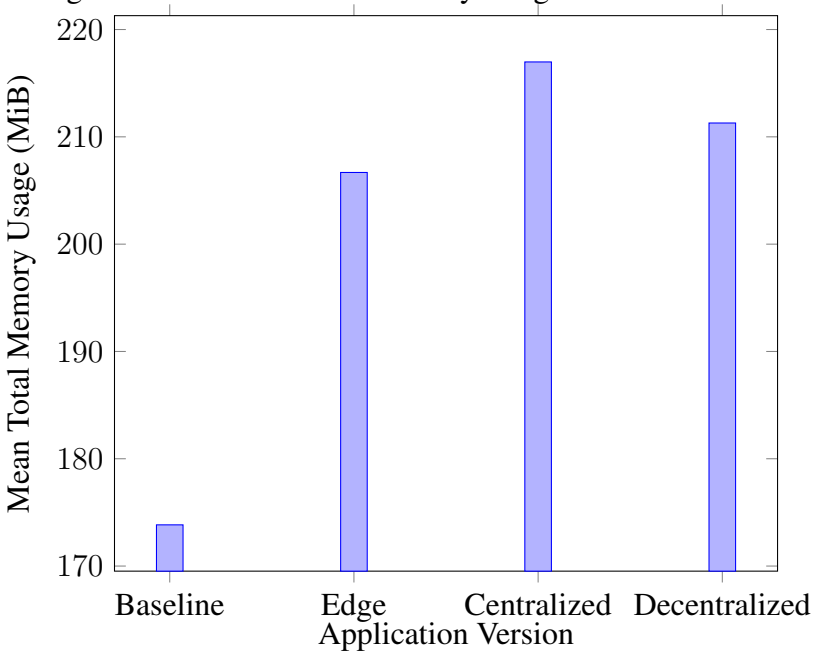


Figure A.11: Mean Total Memory Usage - *transferAmount*

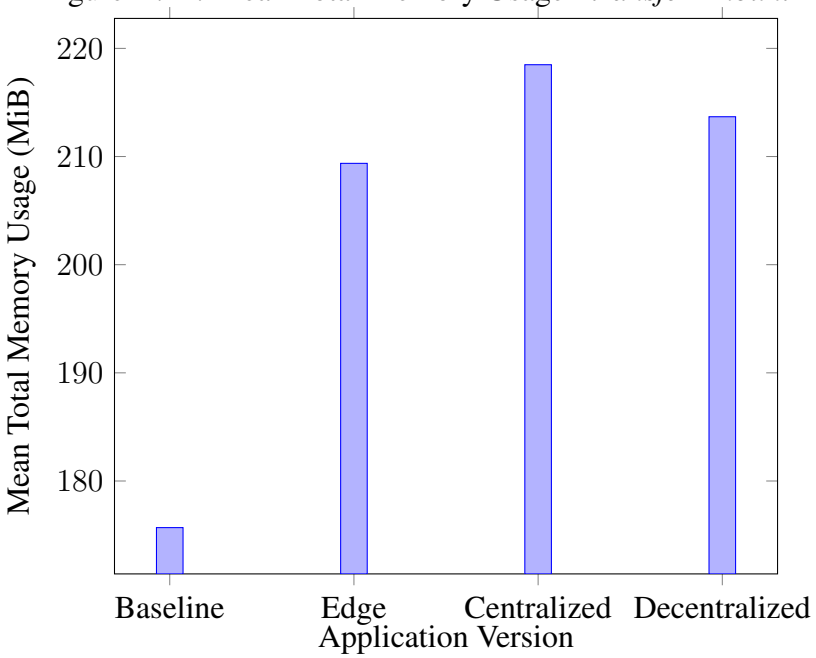


Figure A.12: Mean Total Memory Usage - *transferAmountAndNotify*

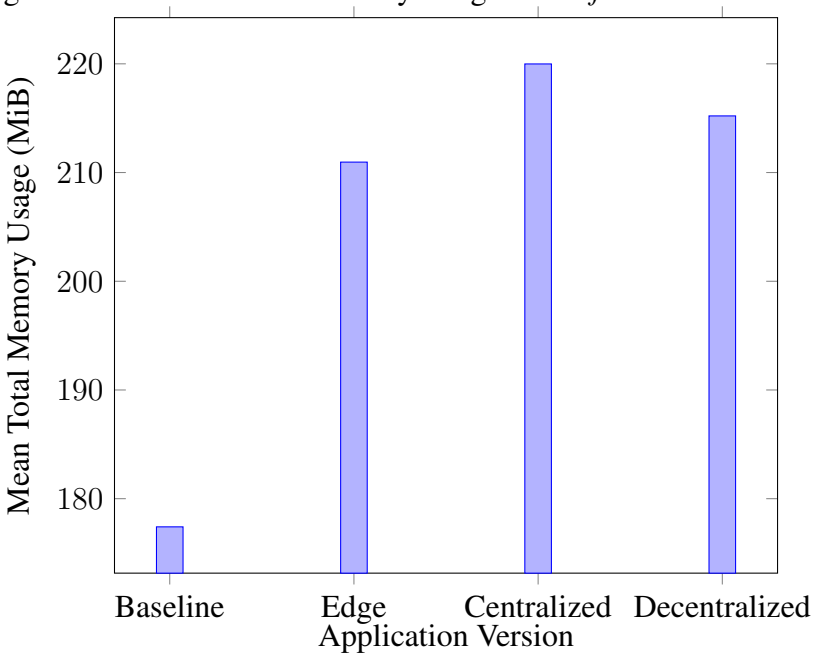


Figure A.13: Mean Total Transmitted Data - *createAccount*

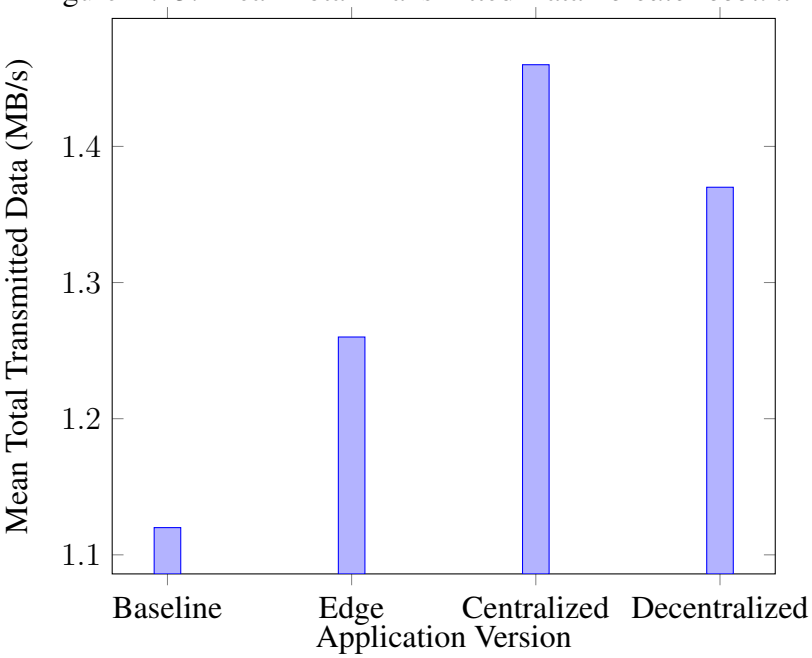


Figure A.14: Mean Total Transmitted Data - *createCustomer*

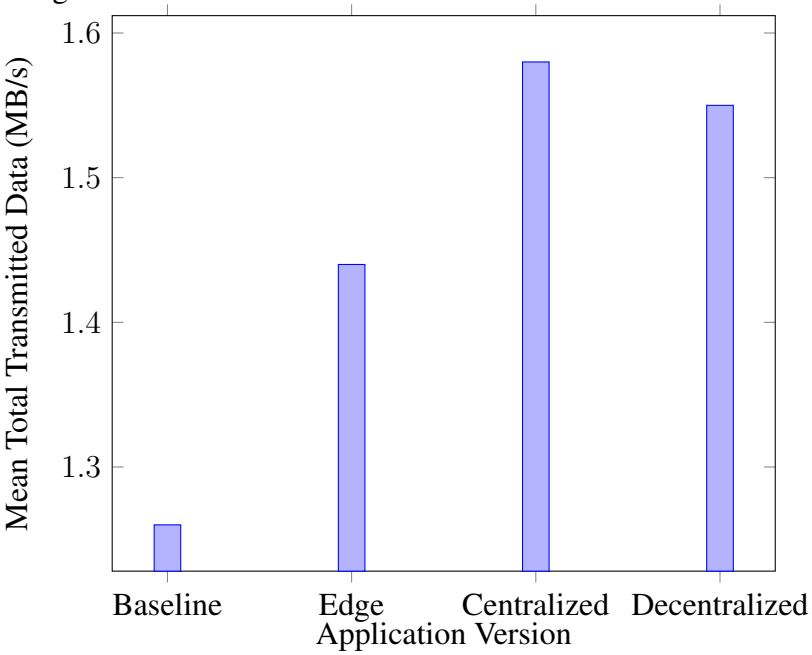


Figure A.15: Mean Total Transmitted Data - *transferAmount*

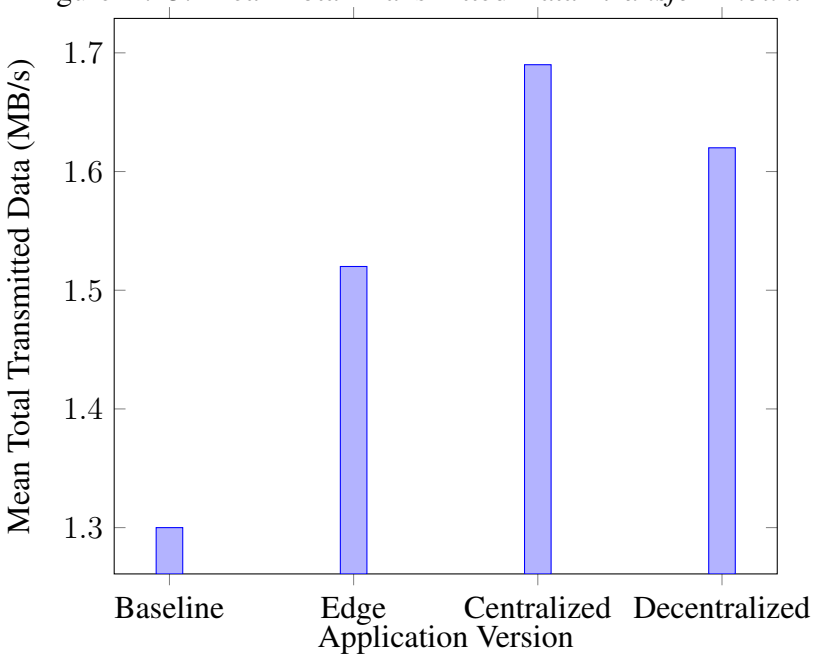


Figure A.16: Mean Total Transmitted Data - *transferAmountAndNotify*

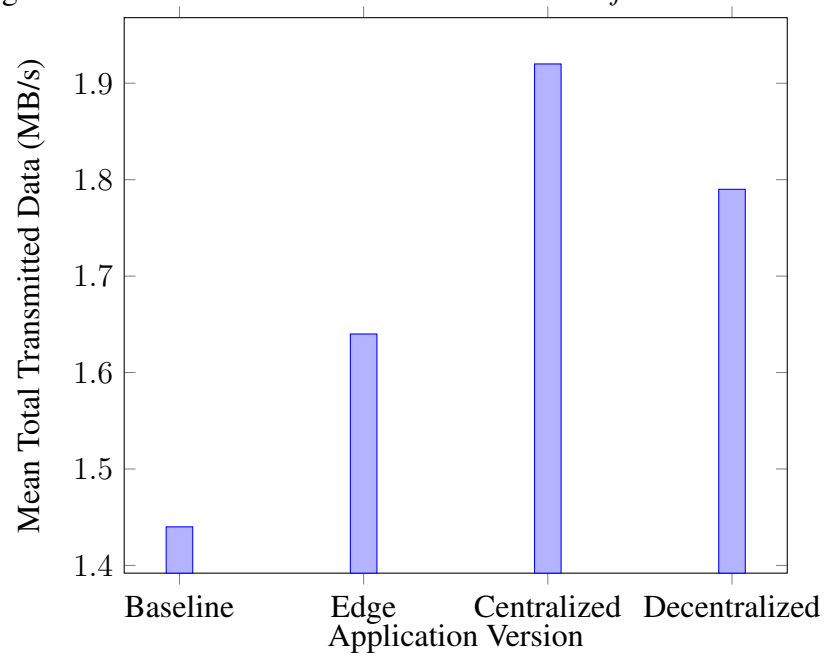


Figure A.17: Total Disk Usage