SCHOOL OF INFORMATICS

Data & Web Science MSc Program

Development of a distributed database for edge devices

by

Pantelis Ypsilantis

A DISSERTATION SUBMITTED TO THE ARISTOTLE UNIVERSITY OF THESSALONIKI FOR THE DEGREE OF MASTER OF SCIENCE IN THE DEPARTMENT OF INFORMATICS

Supervisor Professor: Dr. Anastasios Gounaris

June 2024

Contents

List of Images

List of Tables

Keywords

Edge Computing, Java, Spring, Docker, Kafka, Synchronization, SQLite, Security, JPA, P2P, Peer-To-Peer, DHT, Distributed Hash Table, API

Abstract

In today's rapidly advancing technological landscape, traditional data handling pipelines often struggle with inefficiency. Edge computing is a transformative paradigm offering efficient and decentralized data processing capabilities. It is a distributed computing framework that brings enterprise applications closer to data sources like IoT devices or local edge servers [51], promising benefits such as faster insights, improved response times, and enhanced bandwidth availability. This thesis embarks on a comprehensive exploration of edge computing across six key parts.

Through an analysis of use cases demonstrating the significant benefits of edge computing, we identified a gap in existing distributed databases. Specifically, there was a lack of solutions supporting local storage alongside an embedded decentralized query engine. As a result, we propose a new distributed edge database in this thesis, designed to store data locally while enabling users to access data from other nodes and respecting privacy concerns. During the implementation process, we delve into the fundamental concepts of edge computing, as well as its advantages and disadvantages.

Additionally, our implementation required a thorough understanding of security issues inherent in edge systems. Our analysis led to an extensive description of these issues, supplemented by research into existing databases suitable for our edge database solution.

To provide a comprehensive understanding of our framework, we conducted a detailed analysis of the architecture of our proposed edge database and its sub-components. Central to our novel framework is the utilization of SQLite, enabling multiple instances to be installed on different edge devices, such as smartwatches. Another critical aspect is the integration of distributed hash tables, facilitating distributed queries across the entire cluster, even when data is stored on different devices. This capability is essential for maintaining efficient data retrieval and processing in a distributed network.

To assess the strengths and limitations of our framework, we conducted various test cases and experiments during the implementation phase. The results of these tests highlight areas of robust performance, as well as potential bottlenecks and challenges requiring attention.

In conclusion, we present proposals for optimizing our system and enhancing its robustness, providing a comprehensive overview of our contributions to the field of edge databases.

Acknowledgements

I am profoundly grateful to the numerous individuals who supported me throughout the journey of completing my MSc thesis.

First and foremost, I extend my deepest thanks to my supervisor, Dr. Anastasios Gounaris. His insightful guidance, consistent encouragement, and positive outlook were crucial to the successful completion of this thesis. His unwavering belief in my work from the very beginning provided me with the motivation to persevere through the challenges I encountered.

I would also like to express my sincere appreciation to Theodoros Toliopoulos. His support, expertise in the research topic, and belief in my work were instrumental in achieving this result.

Special thanks go to my family, whose unwavering moral and material support have been a cornerstone of my success. Their confidence in my abilities has been a constant source of strength. I am particularly grateful to my girlfriend, Mary, for her steadfast encouragement and understanding during this demanding period. Her support has been invaluable.

Finally, I extend my heartfelt thanks to my friends and colleagues. Their stimulating discussions and companionship offered much-needed respite from my research. The camaraderie and new experiences we shared significantly enriched this journey, contributing to my personal and professional growth.

Thank you all for your invaluable support and belief in my work. This accomplishment would not have been possible without you.

1 Introduction

In our interconnected world where data generation and consumption are skyrocketing, traditional data processing approaches are encountering new challenges. Edge computing is a paradigm that seeks to revolutionize the data processing landscape by bringing computation and data storage closer to the source. This chapter introduces the concept of edge computing and outlines its pivotal role in addressing latency, enhancing real-time capabilities, and reducing network congestion.

1.1 Motivation

Let's imagine a bustling space station orbiting Earth, where astronauts rely on an intricate network of sensors to monitor environmental conditions crucial for their safety and mission success. Traditionally, data from these sensors is transmitted back to Earth for processing, but this process is slow and prone to delays. Consider a scenario where a critical system malfunction is detected. The time it takes to transmit, process, and relay instructions could escalate into a life-threatening situation for the astronauts. By deploying data processing units directly on the space station, adjacent to the sensors, critical data can be analyzed in real-time without reliance on Earth-based systems. This enables an immediate response to emergencies and enhances mission safety. With edge computing, astronauts gain the capability to receive instant alerts and insights, empowering them to take swift action. By selectively transmitting only essential data back to Earth, bandwidth usage is optimized. Additionally, astronauts might face situations that require data from the past or from another space station. Given that, they should have the ability to easily access the data placed in other edge nodes.

Furthermore, the significance of edge computing is underscored by its application in scenarios closer to home. Take, for instance, the collection of personal health data from smartwatches, where the monitoring of metrics like heart rate demands robust privacy safeguards. In such contexts, users rightfully demand assurances that their sensitive health information remains confidential and firmly under their control. However, there also exists a pressing need to leverage this data for broader insights, such as comparing individual health metrics with anonymized population data or for medical research purposes. Balancing these demands presents a unique challenge, necessitating innovative solutions that allow for data analysis and utilization while upholding stringent privacy standards. This delicate balance underscores the critical role of edge computing in empowering individuals to derive value from their personal data while maintaining privacy and control.

Traditional centralized database architectures present numerous drawbacks, including heavy network loads, dependency on centralized nodes, and inherent data privacy concerns. To address these challenges and harness the benefits of edge computing, our thesis explores the development of a distributed edge database system that empowers users to locally store their data while simultaneously being able to access data from other edge nodes with respect to security for both sides.

This combined use case underscores the versatile applications of edge computing, from enhancing safety in space exploration to safeguarding user privacy in personal health data analysis. It motivates the development of a new edge database system that addresses both the spatial and security needs of modern data processing, ushering in a new era of efficiency and privacy in data management.

1.2 Objectives

Based on the use cases described in the previous section, there arises a compelling need for a distributed edge database system that can facilitate real-time data processing, ensure data privacy, sovereignty, and seamless interaction with other edge nodes within the same network or cloud. As it is imperative for data to be stored locally to achieve these objectives, ensuring efficient synchronization across distributed nodes becomes essential. The investigation conducted into existing distributed database solutions, extensively analyzed in Section 3.2, reveals deficiencies in meeting these requirements. These solutions often lack tailored support for localized data storage and access, instead relying on internal mechanisms that may compromise scalability and privacy. Consequently, there is a pressing need for the development of a bespoke solution capable of addressing the observed shortcomings in current distributed database frameworks, while closely aligning with the principles of edge computing. This solution aims to enhance data privacy, ensure data sovereignty, and optimize synchronization processes, thereby offering a robust and efficient alternative to existing databases. An additional objective is to develop the new edge database with the ability to be easily adaptable to various applications and to incorporate the latest technologies to ensure that the system can be maintained and enhanced effortlessly.

1.3 Contribution

In response to identified shortcomings in existing database systems, we have developed a novel distributed edge database solution. Our database architecture enables local storage of data on each edge node while facilitating data retrieval from other nodes within the network. This functionality is achieved through efficient synchronization algorithms, ensuring seamless communication and data comparison across the network. By decentralizing data storage and processing, our system not only prioritizes data sovereignty and privacy but also empowers users with greater control over their personal data. This initiative not only addresses pressing concerns surrounding data privacy and sovereignty but also represents a significant advancement in the development of robust, user-centric distributed database solutions.

Our system can be divided into two main components: the internal handling of data and the networking between different edge nodes of the cluster.

For the internal handling of data, our proposed edge database is built on top of the SQLite database. To facilitate retrieval and storage of data from/to this node or other nodes, we have implemented a REST API using Spring. Various controllers and endpoints are utilized to perform the required actions quickly and without any intermediate steps.

From a network perspective, to achieve our goals, we leverage peer-to-peer (P2P) architecture for communication between nodes of the cluster and distributed hash tables (DHTs) for efficient synchronization and data management. Both P2P architecture and DHTs are analyzed in Section 4.1. In our system, each node utilizes a distributed hash table to map keys to corresponding data values, enabling efficient lookup and retrieval operations. Additionally, the P2P architecture ensures that nodes can seamlessly communicate and synchronize data without relying on centralized servers or intermediaries. By combining the strengths of P2P architecture and DHTs, our system achieves low latency, high scalability, and robust fault tolerance, while also addressing concerns related to data privacy and sovereignty.

1.4 Structure

This thesis is structured as follows:

Section 2 delves into the details of edge computing, exploring its fundamental concepts, architecture, and advantages and disadvantages.

Section 3 offers a thorough literature review of the multifaceted security challenges introduced by edge computing. Additionally, this section investigates existing databases suitable for edge deployments, evaluating their performance and suitability for our application.

The technical analysis of this thesis is presented in Section 4, which provides a detailed analysis of our proposed system, its sub-components, and the tools/technologies used for the implementation.

In Section 5, we present various test cases and experiments conducted to evaluate the implementation, identifying the system's strengths and limitations.

Section 6 draws conclusions from our findings and provides suggestions for future work, including various new ideas and functionalities that emerged during the deployment of the proposed database.

Finally, there is an appendix section with additional details for each implemented class of our system, along with troubleshooting for potential issues.

2 Background

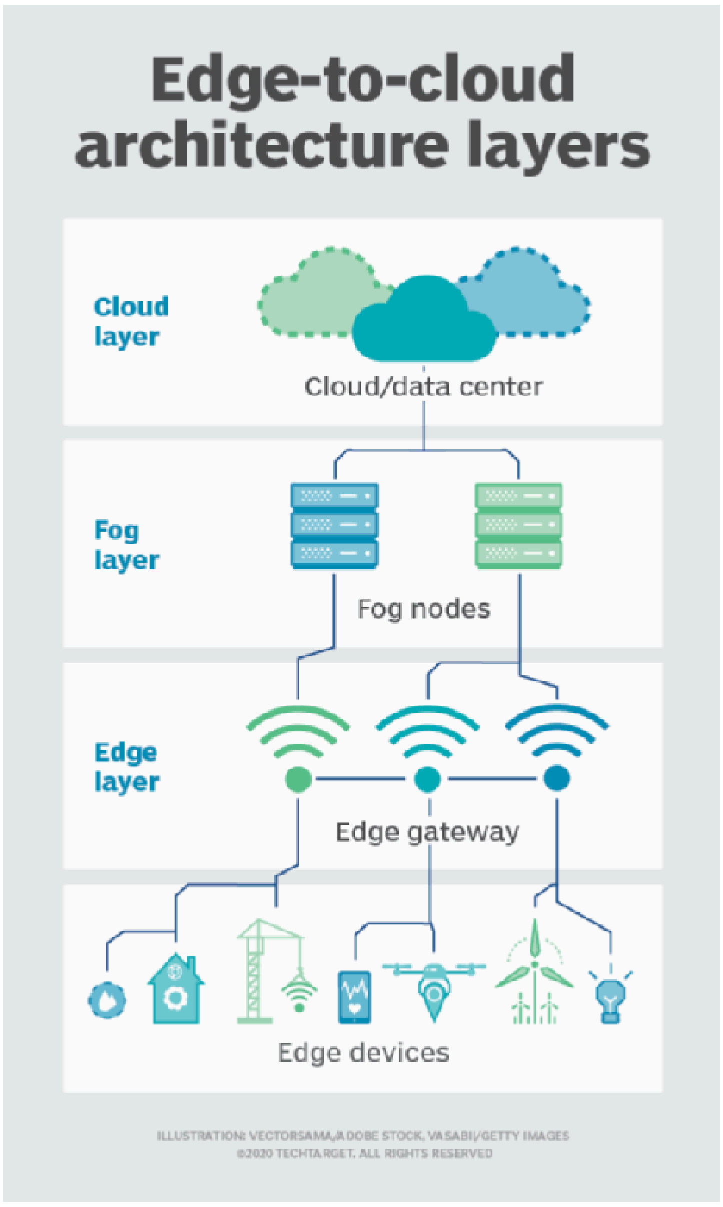
2.1 Introduction To Edge Computing

Before making a deep dive into edge computing terms, let's first understand what edge computing is. The classic pipeline of data storage and processing is composed of 3 major components:

1) The generation of the data which is happening into the edge devices, for example in a receptor
2) The transmission of the data through the network (e.g. internet or LAN) into a cluster or a server in order to be stored
3) The queries for the user, the calculation of the result into the main storage device and the response of the server with the result to the query

Based on those three components there are various issues which came up such as the heavy load of the network or the dependency to the main data storage machine. The idea of edge-computing was created exactly to find a solution to those issues. At the core of edge computing lies the principle of minimizing the physical distance between computational resources and the data source. In simplest terms, edge computing moves some portion of storage and compute resources out of the central data center and closer to the source of the data itself, for example into a smartwatch. Only the result of that computing work at the edge is sent back to the main data center for review and other human interactions. As a result, the overall network load is significantly reduced. By decentralizing data processing, edge computing seeks to optimize response times and enable immediate decision-making. Imagine an autonomous vehicle equipped with edge computing capabilities. Instead of relaying data to a remote server for processing, the vehicle processes data locally, making split-second decisions that can avert accidents or optimize route choices.

Edge computing is closely linked to the principles of cloud and fog computing. While there is a certain degree of intersection between these concepts, they are not synonymous and should typically not be employed interchangeably. It proves beneficial to draw comparisons between these notions and grasp their distinctions. All three concepts are tied to distributed computing and center on the physical arrangement of computing and storage resources concerning the generated data. The key distinction lies in the specific locations of these resources. The following image provides a visualization of the described connections:


Edge-to-cloud architecture layers
Cloud layer
Cloud/data center
Fog layer
Fog nodes
Edge layer
Edge gateway
Edge devices
ILLUSTRATION: VECTORSAMA/ADOBE STOCK, VASABII/GETTY IMAGES
©2020 TECHTARGET. ALL RIGHTS RESERVED

1. Different layers and connections between them [1]

- Edge computing focuses on deploying resources directly at the data source or network edge, making it ideal for tasks requiring low latency and real-time data processing, especially in scenarios with resource limitations.

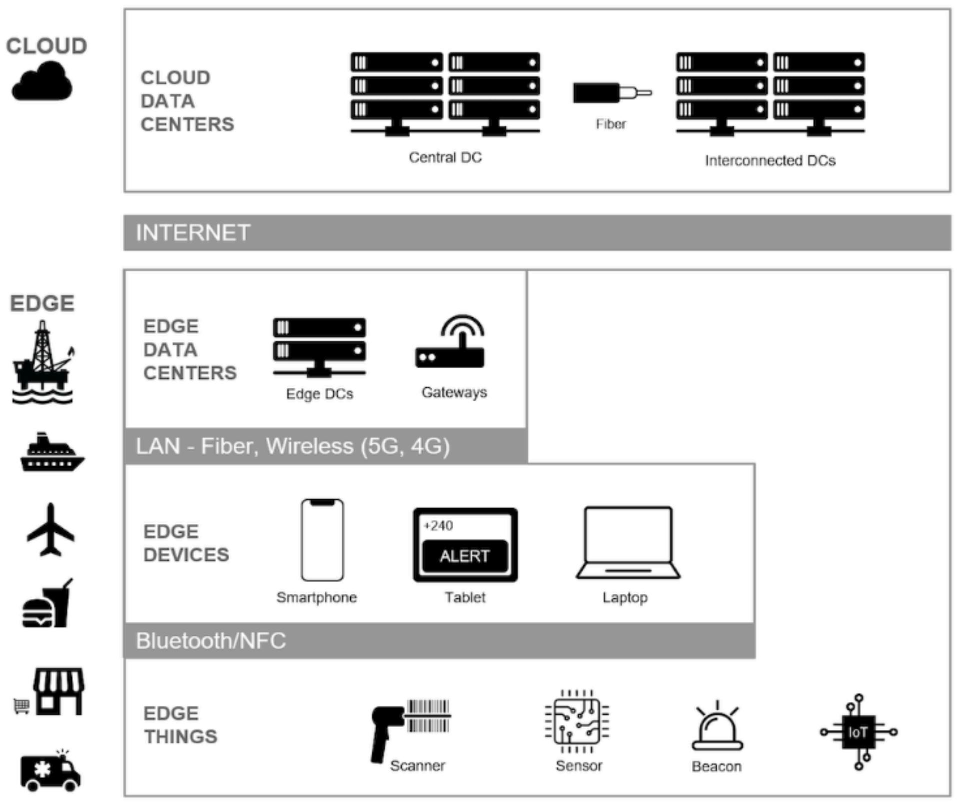
- Cloud computing, on the other hand, relies on centralized data centers located globally, offering unmatched scalability for resource-intensive operations. However, data centers may be situated far from data sources, potentially introducing latency. This makes cloud computing particularly suitable for applications like big data analytics, machine learning, and web services.

- Fog computing takes a middle ground, positioning resources "within" the data, effectively balancing between edge and cloud. It excels in handling extensive data generation across large physical areas, utilizing multiple fog nodes for scalability. This approach proves invaluable for applications such as smart cities and expansive sensor networks.

These distinct characteristics highlight how each computing paradigm addresses data processing and resource deployment, catering to a wide array of use cases and requirements. While each technique presents unique characteristics, they all share a common focus on data processing and management, although their proximity to data sources and scalability options set them apart, making them crucial tools in the modern technological landscape.

2.2 Edge computing Architecture

Edge computing embodies this adaptability by providing a base architecture that prioritizes proximity to data sources, resulting in quicker responses and more efficient data usage, in stark contrast to traditional centralized cloud computing. However, edge computing goes even further by tailoring its architecture to meet the unique demands of various use cases [20, 21], offering custom configurations that take full advantage of its transformative potential. To understand both the foundational and custom layers of edge computing architecture [19, 22], let's explore further. The following image displays the basic architecture of edge computing:



2. Basic architecture of an edge computing deployment [20]

Based on the image, it is easy to define three different layers in the basic architecture:

- **Cloud Layer**: Our architectural canvas begins at the cloud layer, where central and regional data centers form the foundational backbone of this ecosystem. These centers remain pivotal for comprehensive data storage and processing, stepping in when complex computations or extensive data storage surpass the capabilities of the edge layer. It's a symbiotic relationship between cloud and edge.

- **Edge Layer**: Nestled at the heart of the edge computing architecture is the dynamic edge layer. Here, a multitude of strategically positioned edge servers come into play. These servers are the workhorses of real-time data processing, reducing latency to a whisper. Devices within this layer communicate seamlessly with the edge data center, enabling the rapid processing of data at its source, a hallmark feature that distinguishes edge from cloud computing.

- **Device Layer**: The foundation of this architectural masterpiece is the device layer, home to an impressive array of devices. This spectrum encompasses everything from handheld marvels to robust industrial machinery and sensitive sensors. These devices, equipped with sensory prowess, collect an invaluable trove of data, the lifeblood of countless applications. While some devices may sport limitations in computing power, they excel at processing data nearly instantaneously, right at the source.

In line with this fundamental architecture, each edge deployment may feature a slightly distinct structure. One of the primary variations pertains to the choice of database. The chosen database should seamlessly integrate with every layer of the architecture, facilitating smooth data distribution across the ecosystem for instant synchronization. Furthermore, the database should provide support for offline processing, enabling the embedding of databases directly into edge devices. This level of resilience guarantees uninterrupted operations, even when faced with network disruptions. Further details regarding differences among some of the most frequently employed databases will be explored in an upcoming section. Other reasons which are affecting the choice of architecture are connected with the use cases, network conditions, data processing needs, security considerations, scalability requirements, and cost factors. Each architecture is tailored to optimize specific application scenarios, ensuring that edge computing continues to revolutionize the technology landscape in diverse and innovative ways. Some examples of different architectures and be shown below:

- **Centralized Edge**: Suitable for applications with moderate latency requirements, this architecture primarily processes data in regional edge data centers.

- **Distributed Edge**: Ideal for applications demanding ultra-low latency, data processing occurs as close as possible to the source, often at the device level.

- **Hybrid Edge-Cloud**: This approach combines edge and cloud processing based on data nature and application requirements, offering flexibility.

- **Device-Centric Edge**: In device-centric architectures, processing primarily takes place on edge devices, ideal for scenarios requiring rapid local decision-making.

2.3 Advantages of Edge Computing

Edge computing has gained significance due to its ability to effectively tackle emerging network challenges associated with the vast amounts of data generated and consumed by modern organizations. It's not just about the sheer volume of data but it's also about the time-sensitive nature of applications that rely on quick processing and responses.

Take, for instance, the emergence of autonomous drones. These drones will rely on intelligent navigation systems that require real-time data processing and communication. Drones and their navigation systems will need to generate, analyze, and exchange data instantaneously. When you multiply this demand by a substantial number of autonomous drones in the sky, it becomes evident how critical it is to have a fast and responsive network. Edge and fog computing are solutions that address three primary network limitations: bandwidth, latency or reliability. In more details, it is easy to determine the main advantages of edge computing, which are described in detail below:

- **Low Latency**: The latency of a system is the total time which is needed for request and response. Although the networks are constantly growing in terms of speed, for example 5G, the latency could face vast issues when the distance between server and user increases. Those issues could also appear when there are multiple requests from the user (or the application which handles the user requests) or when the database is handling more requests than its capabilities or even when the network is heavily loaded, for example due to an outage. Those delays could have a high impact on the users, especially for cases like autonomous vehicles. Edge computing is solving this issue, along with the issue of network bottlenecks, by moving the database closer to the user and the application. As a result, the system gains the ability to process data instantly and to translate them into immediate actions, enhancing user experiences.

- **Bandwidth Efficiency**: Bandwidth is quantified as the data volume a network can carry per unit of time and it plays a crucial role in network performance. Networks inherently possess finite bandwidth, while wireless communication channels have often more stringent limitations. Issues with bandwidth can severely impact a company's operations, leading to slow data transfer, delayed communication, and reduced productivity. It may result in disrupted services, poor customer experiences, and hindered data-driven decision-making processes, potentially causing financial losses and damage to the company's reputation. Edge computing emerges as a pivotal solution to alleviate the burden on network bandwidth and centralized servers. For instance, in video surveillance systems, implementing edge computing involves processing surveillance footage at the local level, significantly reducing the necessity to transmit substantial data volumes across the network.

- **Offline Capability**: Edge devices are frequently engineered to excel in scenarios with limited or intermittent internet connectivity. This capability proves especially beneficial in environments characterized by unreliable or expensive internet access. By deploying edge devices equipped with ample processing capabilities and storage, businesses can guarantee uninterrupted functionality, even when faced with challenging network conditions. Nevertheless, it's essential to recognize that the degree to which edge computing can function without an internet connection varies based on the unique demands of each application. While certain applications may necessitate periodic connectivity for tasks like updates, data synchronization, or remote management, the overarching benefit of edge computing lies in its capacity to enhance autonomy and resilience in situations marked by intermittent or absent network connectivity.

- **Cost Efficiency**: Edge computing proves highly advantageous in cost minimization efforts. It achieves this through several means, notably by reducing data transmission and centralized storage expenses. By processing data locally at the edge, organizations can curtail the substantial costs associated with transmitting and storing vast amounts of data over long distances to centralized servers or cloud facilities. Additionally, edge computing mitigates the need for continuous, high-bandwidth network connections, further lowering operational expenses. Moreover, the decreased reliance on centralized cloud services often translates into reduced subscription and maintenance costs, contributing to overall cost-efficiency in various industries and applications.

- **Scalability**: Edge computing's distributed architecture offers impressive scalability and adaptability. Unlike traditional centralized data centers, edge devices can be strategically placed across various locations, creating a network of computational resources. This flexibility allows organizations to effortlessly expand processing capabilities as required and distribute workloads to prevent device overload. Scalability in edge computing means easy addition or removal of devices to meet changing requirements, whether to handle increased data generation or support local data processing in new locations.

- **Privacy Enhancement**: Last but not least, edge computing provides a substantial advantage in terms of security and data privacy. Unlike traditional cloud-based models, edge computing enables the local processing of sensitive data, effectively addressing concerns surrounding data security and privacy. This approach aligns with data sovereignty regulations, ensuring that data remains within the boundaries of applicable laws and regulations. Edge devices can process data locally, safeguarding sensitive information before any transmission to remote cloud or data centers, particularly when dealing with diverse jurisdictional requirements. Furthermore, edge security measures enhance data protection by employing encryption for data in transit and fortifying edge deployments against potential security threats. This holistic security and privacy approach is valuable in various sectors where safeguarding sensitive information is a top priority, like personal medical information.

2.4 Disadvantages of Edge Computing

While the advantages of edge computing are undoubtedly enticing, it's crucial to acknowledge the inherent challenges it presents. The primary concern revolves around data consistency, which serves as a significant deterrent for some companies considering the migration to edge computing. Other drawbacks are associated with resource availability, the complexity of implementations, and potential security issues that may arise. Further elaboration on these disadvantages is provided below:

- **Limited Resources**: One notable drawback of edge computing lies in its constrained resource capacity. In contrast to expansive cloud computing environments offering a diverse range of resources and services, edge deployments are typically tailored to specific, predefined purposes, often with limited resources and a restricted service portfolio. Edge devices frequently operate with constrained storage and processing capabilities, which can impede the execution of complex computations, especially when dealing with intricate tasks like machine learning on these resource-limited edge devices. Furthermore, compared to core data centers, edge computing infrastructures frequently lack the comprehensive technical support and infrastructure, contributing to their less robust implementation and necessitating solutions for various technical challenges to ensure efficient operation.

- **Data Consistency**: Achieving and maintaining data consistency in edge computing is a formidable challenge, particularly in dynamic and distributed environments. This challenge becomes evident when considering scenarios like user registration in applications. Ensuring the uniqueness of usernames across various edge nodes is crucial, as it directly impacts user identification and system reliability. In edge computing, data is distributed across multiple locations, making it essential to consult other copies of data to validate the uniqueness of a username. However, this process introduces latency, similar to the challenges faced when centralizing data in a single location. The decentralized nature of edge computing, with data residing on dispersed devices, necessitates intricate synchronization mechanisms to prevent data inconsistencies and ensure uniformity across the edge network.

- **Risk of Data Loss**: Edge computing introduces a heightened risk of data loss. This risk is primarily driven by the nature of edge devices, which are designed for resource optimization and often discard data deemed irrelevant to their immediate tasks. However, this data, although seemingly insignificant at the edge, can hold critical value for broader analytics and decision-making processes. The challenge lies in distinguishing between truly irrelevant data and information that may be valuable for subsequent analysis. Inadvertent data loss at the edge can lead to incomplete datasets and flawed insights. So, ensuring robust data management strategies and carefully defining what data should be retained and what can be safely discarded is essential to mitigate the risk of data loss in edge computing implementations.

- **Security Concerns**: IoT devices are commonly used at the edge. Although, they are regularly vulnerable, necessitating a robust approach to device management, policy-driven configuration enforcement and stringent security measures for computing and storage resources. This includes vigilant software patching, updates, and encryption for data both at rest and in transit. Additionally, edge computing decentralizes data processing, introducing security risks at each localized point within the edge network, as not all edge devices possess equivalent authentication and security capabilities. More details about the security challenges can be found in the next section.

- **Complexity**: Effectively managing a distributed edge system entails handling tasks like synchronization, version control, and fault tolerance, which demand meticulous coordination. Despite the advantages of edge computing in mitigating traditional network limitations, ensuring a minimal level of connectivity remains essential, even in edge deployments intended to be flexible and adaptable. Achieving success in edge computing requires a well-considered design capable of accommodating challenging connectivity conditions, which adds complexity to the implementation. Moreover, the limited lifespan of IoT devices necessitates regular replacements, further complicating matters by requiring the forecasting of potential node failures

3 Literature Review

3.1 Security Challenges In Edge Computing

Current edge computing infrastructures are increasingly susceptible to severe cyber-attacks, potentially resulting in significant financial losses. This topic is extensively analyzed in paper [23]. The prevalence of attacks targeting these infrastructures has seen a remarkable surge in recent years. A notable real-world example is the Mirai virus, which, upon its release in August 2016, swiftly compromised more than 65,000 IoT devices within the initial 20 hours. This exploit primarily targeted the devices' weak authentication vulnerabilities [6]. Subsequently, the compromised devices were harnessed to form botnets, orchestrating DDoS attacks against edge servers and causing the disruption of more than 178,000 domains. The leading root causes of security threats in edge devices can be attributed to the following key factors:

- **Protocol-Level Design Flaws**: Those flaws in edge computing systems often stem from the diverse operating systems and protocols, such as communication protocols, employed by edge devices which differ significantly from the standardized systems found in typical general-purpose computers. The absence of standardized regulation, combined with the creators' tendency to prioritize utility and user experience over security considerations, contributes to inherent design weaknesses. These vulnerabilities create opportunities for attackers to exploit, enabling them to achieve various objectives. These objectives can range from the disruption of regular services, such as DDoS attacks, to the complete takeover of an edge device or server through tactics like malware injection attacks.

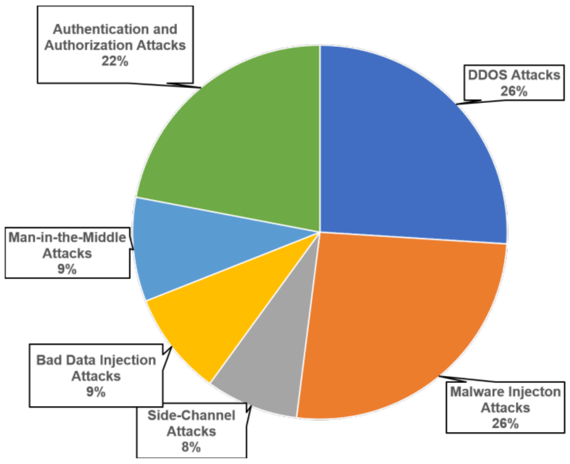
- **Isolated and Passive Defense Mechanisms**: Current edge computing defense mechanisms are isolated and passive. They're isolated because each mechanism may only counter certain attacks, leaving them ineffective against most other threats. They're passive because they rely on predefined rules and lack the ability to autonomously engage in proactive defense. These limitations create a rigid defense framework, leading most existing solutions to adopt a “detect and patch” philosophy, which is often only effective after detecting attacks.

- **Implementation-Level Flaws**: These flaws pertain to errors and vulnerabilities that arise during the practical implementation of edge computing functionalities, potentially undermining the security of a protocol or system, even when theoretical security measures are in place. These vulnerabilities typically stem from two primary sources: misinterpretation of the protocol's core principles during development and challenges encountered when adapting protocols to the context of edge computing, resulting in inconsistencies. Furthermore, security frameworks encounter compatibility issues when transitioning to edge computing systems, given the diverse computational capabilities, various operating systems, network configurations, and communication protocols involved. Additionally, security frameworks designed for specific edge computing applications may not seamlessly adapt to different scenarios due to the diversity of devices and communication protocols. Such implementation flaws can create opportunities for attackers to exploit vulnerabilities and bypass protocol security, including authentication attacks.

- **Code-Level Flaws**: Code constitutes the fundamental component of a program, dictating the precise execution flow that a processor must follow. These vulnerabilities primarily focus on system bugs capable of causing memory failures or corruption, including stack and heap overflows, use-after-free errors involving dangling pointers, and format string vulnerabilities, among others. These vulnerabilities are prevalent even in well-known applications, as exemplified in [7]. Such issues often emerge during the intricate coding process associated with extensive codebases. Exploiting these vulnerabilities equips attackers with a spectrum of potential outcomes, ranging from disrupting regular services to gaining complete control over an edge device or server, as evidenced by malware injection attacks.

- **Lack of Fine-Grained Access Control Models**: Edge computing presents distinct access control challenges due to its intricate systems and diverse applications. Traditional access control models, characterized as “coarse-grained,” typically employ a limited set of basic permissions (No Access, Read Only, Write Only, Read & Write). These models lack the granularity required for precise control over resource access, user actions, and timing, which is a hallmark of “fine-grained” access control. Fine-grained access control offers enhanced security by restricting access to only the necessary resources and actions, thus minimizing potential vulnerabilities. However, the complexity and diversity of edge environments, resource constraints, and the absence of standardization have hindered the widespread implementation of fine-grained access control. This gap leaves edge systems vulnerable to unauthorized access and security threats, highlighting the urgent need for tailored and standardized fine-grained access control solutions to bolster security in edge computing environments.

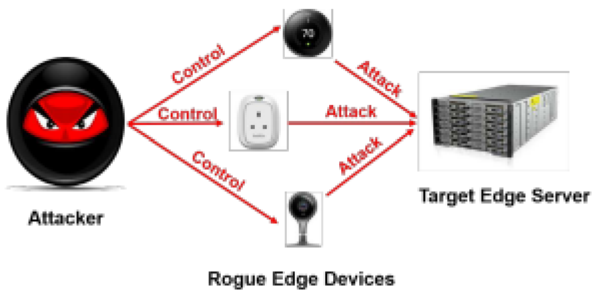
- **Other Causes**: Finally, security weaknesses in edge computing may arise from three other sources. Firstly, the comparatively weaker computational power of edge servers, in contrast to cloud servers, renders them more vulnerable to attacks that might no longer be effective against their cloud counterparts. Similarly, edge devices often have less robust defense mechanisms compared to general-purpose computers, making them susceptible to threats that might not pose significant risks to desktop systems. Furthermore, “Attack Unawareness” presents a challenge, as many IoT devices lack user interfaces, making it difficult for users to detect compromises or ongoing attacks. Lastly, “Data Correlations” introduce another layer of vulnerability, as hidden connections between sensitive and less sensitive data can be exploited by attackers to infer or tamper with the sensitive data, using tactics like side channel attacks or bad data injection attacks. These vulnerabilities collectively underscore the need for robust security measures tailored to the unique aspects of edge computing.



3. Percentages of most common attacks in edge systems [23]

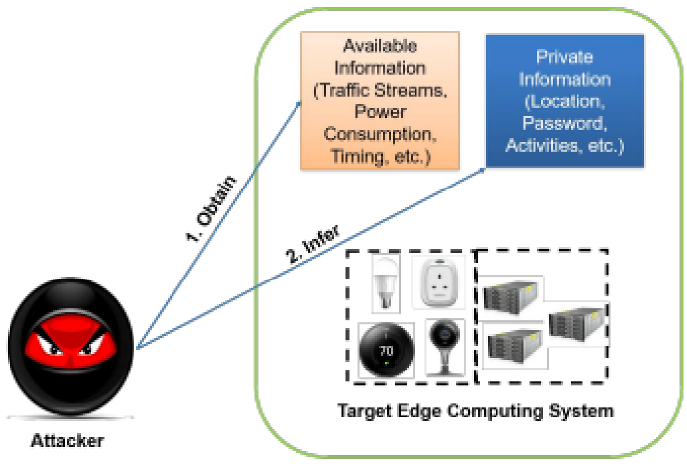
Due to these factors, edge devices, and edge computing in general, are susceptible to numerous security threats. While defensive mechanisms have been implemented for many of these threats, an exploration of them is beyond the scope of this thesis. Nevertheless, some of the most prevalent and impactful attacks are outlined below:

- **DDOS Attacks**: A Distributed Denial of Service (DDoS) attack aims to disrupt server operations using distributed resources, often a botnet of compromised edge devices. Attackers flood a victim with packets or employ malformed packets, depleting its hardware resources and causing service disruptions. DDoS attacks in edge computing include flooding-based attacks and zero-day attacks. Flooding-based attacks involve overwhelming servers with massive amounts of traffic, while zero-day attacks exploit undisclosed vulnerabilities for memory corruption and service disruption, making defense challenging



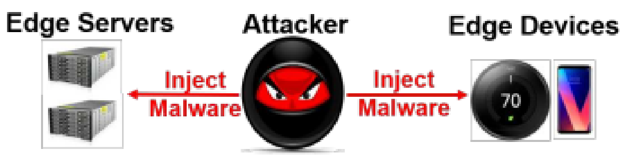
4. Visualization of DDOS Attacks [23]

- **Side Channel Attacks**: In the context of edge computing, those attacks pose a significant threat to user privacy and data security. Those attacks leverage seemingly innocuous, publicly accessible information, which is surreptitiously linked to sensitive data, to compromise security. Edge computing environments offer various avenues for these attacks, making them a versatile tool for malicious actors. Attackers gather side channel information from sources like communication signals, electric power consumption patterns, and data from smartphone embedded sensors. Before delving into the specifics of these attacks, it's essential to categorize them into distinct types based on their methodologies and targets, allowing for a more comprehensive analysis of their mechanisms and potential risks.


Available Information (Traffic Streams, Power Consumption, Timing, etc.)
Private Information (Location, Password, Activities, etc.)
1. Obtain
2. Infer
70
Target Edge Computing System
Attacker

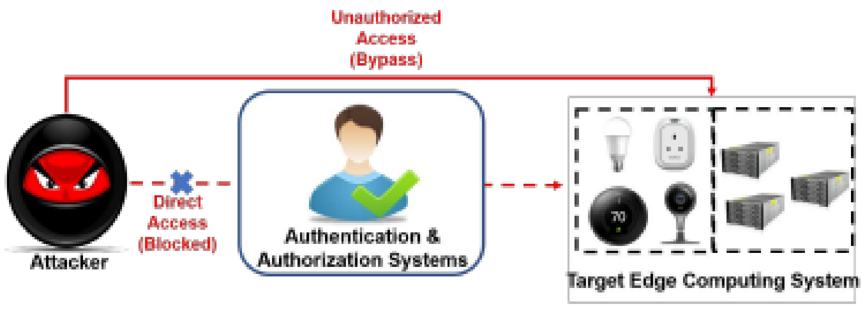
5. Visualization of Side Channel Attacks [23]

- **Malware Injection Attacks**: This type of attacks is a significant menace in computing systems, posing severe risks to data integrity and system security. Unlike traditional Internet or general-purpose computer infrastructures with robust computational resources, edge devices and low-level edge servers often lack adequate protection, making them susceptible to malware injection. These attacks can be broadly categorized into two types: server-side injections, which target edge servers, and device-side injections, focused on edge devices. Server-side injections encompass various attack subtypes, including SQL injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Server-Side Request Forgery (SSRF), and XML Signature Wrapping. Device-side injections exploit vulnerabilities in heterogeneous IoT devices, often relying on zero-day vulnerabilities for remote code execution or command injection, making them potent threats to edge computing environments.



6. Visualization of Malware Injection Attacks [23]

- **Authentication and Authorization Attacks**: Authentication and authorization processes are critical components of security across various computing environments. Authentication verifies the identities of users or entities requesting access to specific services, ensuring that only legitimate individuals gain entry. In contrast, authorization defines access rights and privileges, ensuring that entities operate within their designated boundaries and don't exceed their permitted actions. In edge computing, authentication primarily occurs between edge devices and servers, confirming the legitimacy of requests and interactions. Authorization involves granting permissions by edge servers to devices or applications, and occasionally devices or applications granting permissions among themselves, such as in home automation systems. These attacks are categorized into four types: dictionary attacks, which involve systematically trying various credentials to gain unauthorized access; exploits of authentication protocol vulnerabilities, where attackers exploit flaws in the authentication process; exploits of authorization protocol vulnerabilities, where attackers manipulate weaknesses in the authorization process; and overprivileged attacks, in which entities are granted excessive access rights, potentially leading to misuse or unauthorized actions, such as unauthorized access or privilege escalation. Each type poses unique threats to edge computing security.



7. Visualization of Authentication & Authorization Attacks [23]

3.2 Databases for edge computing

Databases play a pivotal role in the landscape of edge computing, where the need for localized data storage, quick access, and resilience to intermittent connectivity is paramount. Given that, it is crucial to delve into a spectrum of databases specifically tailored for edge devices, ranging from lightweight solutions ideal for resource-constrained environments to robust, globally distributed databases designed to meet the demands of modern edge computing. These databases empower edge devices to store, retrieve, and process data efficiently, facilitating real-time decision-making and enhancing the overall performance and reliability of edge applications. Each database offers a unique set of advantages and trade-offs, making them suitable for a variety of edge computing scenarios. After a comprehensive investigation, the following databases were selected as the most suitable for edge computing tasks:

- **SQLite** [8]: SQLite is a small, fast, self-contained, high-reliability, full-featured, SQL database engine. It is the most used database engine globally, built into mobile phones, computers, and countless applications. It is commonly used in embedded systems, mobile applications, and edge IoT devices. As SQLite is a lightweight database, it is ideal for resource-constrained edge devices. Also, it operates without the need for a separate server, simplifying deployment on edge nodes and it supports transactions, ensuring data integrity even in edge environments. On the other hand, it may not perform well under high concurrent write workloads and scaling SQLite horizontally across multiple edge nodes can be complex. Also, it's not optimized for complex query processing. Although, as we have already mentioned in the first section of this thesis, we used SQLite as it fits well in the requirements of our framework. It is lightweight, it provides the basic functionalities which are required and its configuration is very simple and easy to be performed in edge devices.

- **Firebase Realtime Database** [10]: The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and synchronized in real-time to every connected client, allowing edge devices to stay up to date. It supports offline data access, crucial for edge devices with intermittent connectivity. It scales automatically to handle varying workloads in edge environments. However, costs can increase with usage, especially in high-throughput scenarios. Firebase ties you to Google's ecosystem, which may not be suitable for all edge scenarios, and its querying capabilities may not be as flexible. Firebase is commonly used for real-time data synchronization in mobile and web apps, edge IoT applications requiring real-time synchronization and offline support, and collaborative tools that need real-time updates. For example, it's used in an edge IoT application for real-time monitoring of environmental sensors in remote locations as in this case we do not have to store data on the edge but to handle them from edge devices. Overall, Firebase is suitable for edge computing tasks, although it is not suitable for our application as it is cloud-hosted and does not support store of data on the edge nodes.

- **Apache Cassandra** [11]: Apache Cassandra is an open-source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Apache Cassandra is designed for horizontal scalability, handling large datasets and high write loads. However, setting up and configuring Cassandra for optimal performance can be complex, and it can be resource-intensive, unsuitable for highly constrained edge devices. Also, it uses CQL (Cassandra Query Language), which might require a learning curve. As it comes to the usages of Cassandra, it is used for storing large volumes of IoT data generated at the edge, edge content delivery requiring low-latency access, and collecting and analyzing log and event data generated by edge devices. For instance, it's employed in an edge content delivery network for low-latency video streaming. So overall, the Cassandra database is not suitable for our application.

- **Amazon DynamoDB** [12]: Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed for high-performance applications at any scale. DynamoDB comes with a range of features, including built-in security, continuous backups, automated multi-Region replication, in-memory caching, and data import and export tools. In edge environments, Amazon DynamoDB simplifies operational tasks and can automatically scale to accommodate varying workloads. Its capability to provide low-latency data access is crucial for edge applications, but it's worth noting that costs may increase with usage, especially in high-throughput scenarios. DynamoDB, being tied to AWS's ecosystem, may not be suitable for all edge scenarios, and its querying capabilities may have limitations. Common use cases for DynamoDB in edge computing include building serverless edge applications with minimal operational overhead, and supporting real-time analytics for scenarios requiring immediate data processing at the edge. It also functions as a caching layer for frequently accessed content in edge networks. For instance, DynamoDB is employed in a serverless edge application to enable real-time location tracking of delivery vehicles. Based on the features of the DynamoDb, and especially due to the serverless functionality, it is not suitable for our application.

- **Microsoft Azure Cosmos DB** [13]: Azure Cosmos DB is a fully managed NoSQL and relational database tailored for modern application development. It boasts single-digit millisecond response times, automatic and instant scalability, and guarantees speed at any scale. Business continuity is assured thanks to SLA-backed availability and robust security measures. Azure Cosmos DB offers the flexibility of supporting multiple data models, making it ideal for diverse edge data scenarios. Its global distribution capabilities ensure low-latency access to data from edge locations. However, setting up and configuring Cosmos DB, especially for global deployments, can be complex. Costs may also be a concern for large-scale edge deployments due to its pricing model, and integration with Azure services may result in vendor lock-in. This database finds application in storing and processing data from IoT edge devices, building global edge applications with low-latency data access, and handling geo-distributed content from edge locations. For instance, it plays a vital role in a global edge application focused on real-time stock market data analysis. Concluding, CosmosDB is a cloud database that makes it unsuitable for our application as we need to save the data in the edge device.

- **EdgeDB** [14, 15]: EdgeDB is an open-source database designed as a successor to SQL and the relational paradigm, addressing challenging design issues in existing databases. It offers strong typing, data validation, and schema evolution for enhanced data consistency and adaptability over time. Specifically efficient for time-series in edge computing, EdgeDB optimizes data operations using methods like TPEI and TMTree, resulting in improved insert and write throughput, reduced query latency, and lower memory usage when compared to BTrDB. This database excels at merging timeseries streams on edge nodes, employing I/O-friendly data structures, and supporting both local and global queries in collaboration with centralized cloud databases. While its simplicity and developer-friendly features facilitate ease of use, EdgeDB's relatively recent emergence may limit available resources and community support, potentially posing scalability challenges for extensive edge deployments. EdgeDB finds suitability in IoT applications emphasizing data consistency and schema evolution, as well as scenarios demanding content management and data integrity at the edge. Notably, it serves as a robust backend database for edge applications prioritizing data integrity, exemplified by its use in an edge IoT application for managing and analyzing agricultural machinery sensor data. Overall, this database seems promising as a standalone database and we can use it similarly to SQLite. However, the cloud version, which allows us to query both local and global data, does not have a free option and, due to its high price, cannot be used in our target use cases.

- **CockroachDB** [16]: CockroachDB, the world's advanced cloud SQL database, offers effortless scalability, robust resilience, and low-latency performance, regardless of user locations. It prioritizes strong consistency, vital for data integrity, and delivers a distributed SQL database that scales effectively in edge deployments. CockroachDB's failure resilience ensures high availability in edge networks, although setting up and managing clusters can be intricate. Resource demands may pose concerns for edge devices, and both administrators and developers may encounter a learning curve. This database excels in constructing geographically dispersed applications, demanding robust data consistency, scenarios necessitating financial data processing and strong consistency at the edge, and environments spanning multiple clouds, ensuring data integrity throughout. Notably, it plays a role in a geographically distributed edge application, facilitating real-time order processing within a retail chain. So based on the above description, the CockroachDb is not suitable for our application as it is cloud based and we might face issues due to resource demands in the edge devices.

- **RocksDB** [17]: RocksDB, a high-performance embedded database, specializes in key-value data storage. Derived from Google's LevelDB, it's meticulously crafted to harness the capabilities of multi-core processors (CPUs) and leverage fast storage options like solid-state drives (SSD) for I/O-intensive workloads. Built on the foundation of a log-structured merge-tree (LSM tree) data structure, RocksDB excels in high-speed data storage and retrieval, making it a prime choice for edge devices. It boasts persistent storage features, ensuring data durability, yet primarily functions as a single-node database, potentially limiting horizontal scalability. Integrating RocksDB into an application may entail development efforts, and its suitability for intricate queries may trail behind some other database alternatives. RocksDB is frequently adopted in scenarios where high-performance local storage is essential for edge devices, environments where IoT data necessitates local storage and processing, and as a caching or session storage solution for edge applications. Notably, it plays a vital role in edge devices by caching frequently accessed images in a content delivery application. Overall, RockDB seems to be a promising solution for our application.

- **Apache Ignite** [18]: Apache Ignite is a distributed in-memory database and caching platform that finds relevance in the realm of edge computing. It boasts impressive advantages, including in-memory speed for lightning-fast data access, scalability to handle growing edge workloads, data replication for reliability, a distributed SQL query engine for complex queries, and support for stream processing and real-time analytics. However, it comes with some caveats. Apache Ignite can be resource-intensive, making it less suitable for resource-constrained edge devices, and setting up and managing distributed clusters can be complex. Nevertheless, it shines in use cases such as IoT data processing, real-time edge analytics, local edge caching, connected vehicle applications, and retail edge scenarios, where its real-time capabilities and scalability are invaluable for enhancing edge computing performance and responsiveness. Previous versions of Apache Ignite were including a feature which would achieve our goal fast and easy. Although, this feature has been removed in the latest releases so we can use ignite only as a local database, which will limit its potential. So, Apache Ignite cannot be used for our application.

- **VergeDB** [9]: John Paparrizos et al. have proposed a new database named VergeDB in order to overcome the issues which exist with the current databases regarding edge computing. VergeDB is a database designed to handle the massive scale of time-series data generated by Internet of Things (IoT) applications. It focuses on adaptive and task-aware compression of IoT data, making it suitable for edge devices. VergeDB serves as both a lightweight storage engine and an edge-based database that manages compression and in-situ analytics on raw and compressed data. It optimizes resource usage, storage, and network bandwidth to maximize throughput, data compression, and downstream task accuracy. VergeDB offers several advantages. It excels in adaptive compression, optimizing data storage by adjusting compression techniques based on available resources and downstream task requirements. This task-aware database aligns data compression and filtering decisions with specific task objectives, supporting advanced analytics such as anomaly detection, regression, clustering, and classification. VergeDB also offers subsampling capabilities, reducing data volume while retaining essential information, and preserves time-series similarities, facilitating data mining and machine learning tasks. Furthermore, its scalability efficiently handles large volumes of IoT data, reducing the strain on centralized IoT solutions. However, it does come with its share of disadvantages. Depending on the chosen compression method, there may be computational overhead for compression and decompression, and resource-intensive operations, particularly those involving floating-point precision, might not be suitable for edge devices with limited resources. Users may encounter a learning curve when configuring VergeDB's compression settings, and certain compression methods could impact query performance by requiring data decompression before execution. It's important to note that VergeDB is still in the experimental phase, lacking a straightforward installation process, and its stability for production use cannot be guaranteed. Although this database seems to have a lot of potential, there are no available installations and the development of this database is still at a very mature level. Given that, we won't proceed with this database.

- **GhostDb** [47]: GhostDB is a distributed, in-memory key-value database and caching platform that delivers unparalleled performance for dynamic web applications and API-driven services. It excels in providing lightning-fast data access and scalability, making it an indispensable tool for performance-critical applications. Leveraging in-memory storage for exceptional speed in high-throughput applications, it handles growing workloads across distributed environments. Additionally, it ensures high availability and data integrity with robust replication mechanisms, maintaining data consistency and fault tolerance across distributed clusters. GhostDB empowers users with a distributed SQL query engine for efficient querying and supports real-time analytics and stream processing for dynamic data analysis. However, GhostDB requires significant resources due to its in-memory nature, potentially challenging in resource-constrained environments. Setting up and managing distributed clusters may require expertise and entail administrative overhead. Potential use cases include IoT Data Processing, enabling real-time processing of large volumes of IoT data for rapid insights, Real-time Edge Analytics for immediate data analysis and response in edge computing scenarios, and Local Edge Caching to enhance application responsiveness and reduce latency by caching frequently accessed data locally. Based on the details gathered, GhostDB appears to be abandoned, with its last update being three years ago. Additionally, our implementation will be in Java, and GhostDB lacks a Java API. The in-memory feature might lead to performance issues in devices with limited memory. Furthermore, it's unclear how GhostDB handles new entries and where it saves the data. This information is crucial to determine whether GhostDB can be used like a distributed hash table to speed up our application and avoid potential bottlenecks.

- **EventQL** [48]: EventQL is a distributed, columnar database optimized for large-scale data collection and analytics tasks. It features automatic partitioning for seamless scalability, idempotent writes for exactly-once ingestion from streaming sources, and a compact, columnar storage engine for efficient analytics. It offers standard SQL support with automatic parallelization and scalability to petabytes with distributed query execution. Low-latency operations for streaming data, support for timeseries, relational, and key-value data, and an HTTP API with a native TCP-based protocol facilitate easy integration. Fast range scans ensure efficient data retrieval, while a hardware-efficient implementation ensures maximal performance. Its highly fault-tolerant architecture has no single point of failure, and its self-contained setup requires minimal dependencies. EventQL is suitable for storage and analysis of streaming event, timeseries, or relational data, high-volume event and sensor data logging, and joining and correlating timeseries data with relational tables. Although, it is optimized for real-time data analytics processing (OLAP) tasks and may not be suitable for most transactional (OLTP) workloads like our application. Additionally, it does not seem to have a way to set the destination of the data while we need to save the generated data to the source node. Finally, the project seems to be abandoned, as its last update was seven years ago. To sum up, the EventQL database does not fit with the requirements of this thesis.

- **Citus Data** [49]: Citus Data is a PostgreSQL plugin which offers a distributed database solution that enables the scaling of PostgreSQL by distributing data and queries. It provides the flexibility to start with a single Citus node and seamlessly add nodes and rebalance shards as requirements grow, making it a scalable solution for evolving data needs.

Leveraging parallelism, Citus accelerates queries by up to 20x to 300x or more, utilizing techniques such as in-memory data caching, higher I/O bandwidth, and columnar compression for optimized performance. Citus maintains compatibility with the latest PostgreSQL versions, allowing users to leverage their existing SQL toolset and PostgreSQL expertise. This integration ensures familiarity and ease of adoption, minimizing the learning curve for users transitioning to a distributed environment. Additionally, it simplifies the database architecture by consolidating transactional and analytical workloads into a single database, reducing infrastructure complexity and management overhead. This consolidation streamlines operations and facilitates a more efficient use of resources. The Citus database empowers users with distributed tables, providing the scalability needed to support growing datasets and workloads. With the introduction of schema-based sharding in Citus 12, users can seamlessly onboard existing applications with minimal changes and accommodate new workloads such as microservices, further expanding the platform's versatility. From a single node to a distributed cluster, Citus offers the full functionality and capabilities of PostgreSQL, ensuring compatibility and continuity for users across all scales of deployment. Its distributed architecture, combined with the power of PostgreSQL, makes Citus Data a compelling solution for organizations seeking scalable and high-performance database solutions. Citus Data appears to offer a promising solution for edge applications, yet it also presents some drawbacks. The integration of PostgreSQL with the plugin may consume a significant portion of the storage space on our edge device, as they are not particularly lightweight. Additionally, due to the nature of distributed tables, every user would have full access to the data of other users, which is often an undesirable feature for applications handling sensitive data. Furthermore, Citus Data does not clearly specify whether the data are saved in the source node of the generated data or if they are distributed evenly among the nodes. Considering these factors, Citus Data may not be suitable for our application.

- **TiDB** [50]: TiDB is an open-source distributed SQL database, offers a robust solution for Hybrid Transactional and Analytical Processing (HTAP) workloads, boasting MySQL compatibility and horizontal scalability. Its architecture, separating computing from storage, ensures seamless scaling with high availability and strong consistency. Notably, it provides high availability through data replication and the Multi-Raft protocol, enabling geographic redundancy and configurable disaster tolerance. Real-time HTAP capabilities are enhanced by TiDB's dual storage engines: TiKV for row-based storage and TiFlash for columnar storage. This enables simultaneous handling of transactional and analytical workloads with consistent data replication. TiDB's cloud-native design ensures flexible scalability, reliability, and security in cloud environments, appealing to existing MySQL users due to its compatibility. Use cases for TiDB span various industries, including finance, high-concurrency environments, real-time data processing, and data aggregation scenarios. Its distributed architecture, coupled with features like MVCC and distributed ACID transactions, ensures reliability, scalability, and data consistency across environments. Also, TiDB utilizes RocksDB for local storage, providing a high-performance, reliable storage engine for persistent data storage and management within the distributed database system.

RocksDB ensures efficient data storage and retrieval, contributing to TiDB's overall performance and reliability in handling large-scale workloads. Additionally, security is typically managed at the cluster level rather than on a per-node basis. However, it is possible to implement different security measures for specific nodes by utilizing network security protocols, access control lists (ACLs), and encryption mechanisms. For example, enforcing network segmentation to restrict access to sensitive nodes, configure firewall rules to control inbound and outbound traffic, and implement encryption at rest and in transit to protect data integrity and confidentiality. These are some ways to implement additional security features. Moreover, integrating TiDB with external authentication and authorization systems to enforce fine-grained access control policies based on user roles and permissions is a good way to enhance security of the data. Based on the nature and the functionalities of the TiDB, installing it on edge devices may impose resource requirements depending on the device's specifications and the desired configuration of TiDB. While TiDB itself is designed to be lightweight and scalable, running a distributed database system on edge devices with limited resources may present challenges. Edge devices with constrained processing power, memory, or disk space may struggle to handle TiDB's demands, especially in scenarios with high data throughput or complex queries. Finally, TiDB does not make it clear if we are able to use it in order to perform storage-data-on-the-edge and further testing is required for this case.

Based on the results of our investigation, the conclusion is that the databases which can be used are SQLite, EdgeDB and RockDB. For the scope of this thesis, SQLite database will be used as the main target is to build the edge system on top of devices with very limited resources and without the need for complex queries or heavy activities like machine learning.

4 Proposed Framework

In this section, we will outline the structure of the framework in detail. To provide a comprehensive understanding of the implementation, we will begin by defining key terms and concepts related to the tools and technologies utilized. This foundational knowledge will aid in grasping the subsequent discussions on the framework's architecture and functionality.

4.1 Basic Terms

4.1.1 Docker

Docker [39], revolutionizing software development and deployment, serves as an open platform for creating, shipping, and running applications. By decoupling applications from infrastructure, Docker enables rapid software delivery and scalability, leveraging containerization technology to package and execute applications in isolated environments known as containers. These containers encapsulate all necessary dependencies and configurations, ensuring consistent performance across diverse environments.

Docker offers several key features:

- **Containerization**: Docker's containerization technology enables the packaging and execution of applications in lightweight, isolated containers, ensuring enhanced security and isolation while allowing multiple containers to run concurrently on a single host.
- **Portability**: Docker ensures high portability of workloads, allowing applications to operate seamlessly across various environments, from local laptops to cloud providers.
- **Streamlined Development Lifecycle**: Docker facilitates continuous integration and continuous delivery (CI/CD) workflows by enabling developers to work in standardized environments using local containers, ensuring consistency and efficiency throughout the development process.
- **Dynamic Workload Management**: Docker's lightweight and portable nature enables dynamic workload management, allowing for the scaling up or tearing down of applications and services in near real-time to meet evolving business demands.
- **Cost-Effectiveness**: Docker offers a cost-effective alternative to traditional hypervisor-based virtual machines, optimizing server capacity utilization and resource efficiency, particularly in high-density environments and small to medium deployments.

Docker follows a client-server architecture, where the Docker client communicates with the Docker daemon responsible for building, running, and distributing Docker containers. The Docker client and daemon interact via a REST API, facilitating seamless container management. Additionally, Docker introduces several core objects, including images, containers, networks, volumes, and plugins, each playing a crucial role in the Docker ecosystem:

- **Image**: A Docker image serves as a read-only template containing instructions for creating a Docker container. Dockerfiles define the steps needed to create the image and run it, specifying dependencies, environment variables, and commands during the build process.
- **Container**: A container represents a runnable instance of a Docker image. Containers can be created, started, stopped, and deleted using Docker's API or CLI. Docker Compose simplifies the management of multi-container applications, allowing users to define and run them using a YAML file.
- **Network**: Docker networks provide communication between Docker containers, enabling interaction with each other and external networks. They offer isolation and security while maintaining network segmentation and access control.
- **Volume**: Docker volumes facilitate persistent data storage, allowing data sharing between Docker containers and the host machine. They ensure data persistence and portability across container instances, offering flexibility and scalability independently of the container lifecycle.

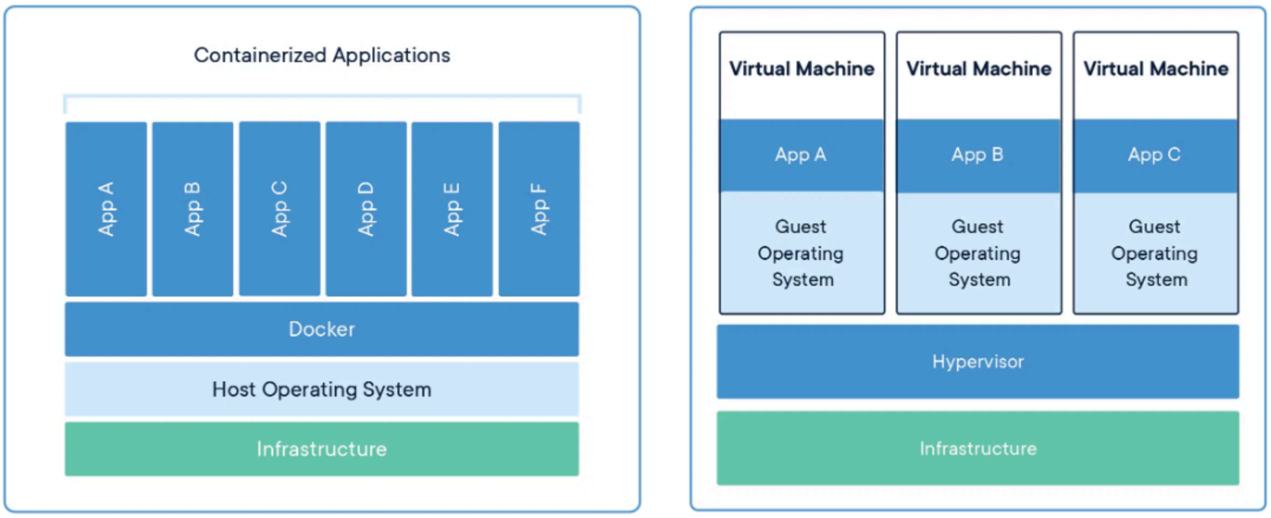
4.1.1.1 Docker containers vs Vms

Containers and virtual machines (VMs) [40] offer similar advantages in terms of resource isolation and allocation, yet they differ significantly in their underlying architecture and functionality. While VMs virtualize hardware, containers virtualize the operating system, resulting in distinct characteristics and use cases for each.

Containers, functioning as an abstraction at the application layer, bundle code and dependencies together. Multiple containers can operate on the same machine, sharing the OS kernel with other containers, and run as isolated processes in user space. Typically, container images are compact, often tens of MBs in size, making them highly portable and efficient. They require fewer resources compared to VMs, allowing for the deployment of more applications without the overhead of additional operating systems.

On the other hand, VMs abstract physical hardware, transforming a single server into multiple virtual servers. A hypervisor enables the simultaneous execution of multiple VMs on a single machine. Each VM includes a complete copy of an operating system, along with the application, binaries, and libraries, resulting in larger disk space requirements, often tens of GBs per VM. Additionally, VMs may have slower boot times due to the need to initialize the entire operating system stack.

In summary, while both containers and VMs offer benefits such as resource isolation, containers excel in terms of portability, efficiency, and scalability, making them ideal for modern application deployment scenarios. VMs, while providing strong isolation, are typically more resource-intensive and may be better suited for scenarios requiring complete virtualization of hardware resources.



8. Comparison of Docker Containers with Virtual Machines

4.1.2 Kafka

Apache Kafka [41, 42] stands as an open-source distributed event streaming platform, serving as a cornerstone for numerous companies in facilitating high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Kafka has emerged as a game-changer in modern data processing, providing an efficient and scalable solution for handling real-time data streams. Kafka's architecture is built around the publish-subscribe messaging pattern, where producers publish streams of events to topics, and consumers subscribe to those topics to process the events. This decoupling of data producers and consumers enables Kafka to handle massive volumes of data with low latency, making it ideal for use cases such as log aggregation, real-time analytics, monitoring, and messaging. Some key features of Kafka are listed below:

- **High Throughput**: Kafka enables the delivery of messages at network-limited throughput through a cluster of machines, achieving latencies as low as 2ms. This ensures that data can be processed and delivered rapidly, meeting the demands of real-time applications and analytics.
- **Scalability**: Kafka offers scalability to production clusters, accommodating up to a thousand brokers, trillions of messages per day, petabytes of data, and hundreds of thousands of partitions. Kafka clusters can elastically expand and contract storage and processing as required, ensuring seamless scalability to handle growing data volumes.
- **Permanent Storage**: Kafka provides the capability to store streams of data securely in a distributed, durable, fault-tolerant cluster. This ensures that data is reliably persisted and can be accessed for batch processing, historical analysis, or replaying events.
- **High Availability**: Kafka efficiently stretches clusters over availability zones or connects separate clusters across geographic regions, ensuring high availability and fault tolerance. This enables Kafka to maintain continuous operation even in the face of hardware failures or network partitions.

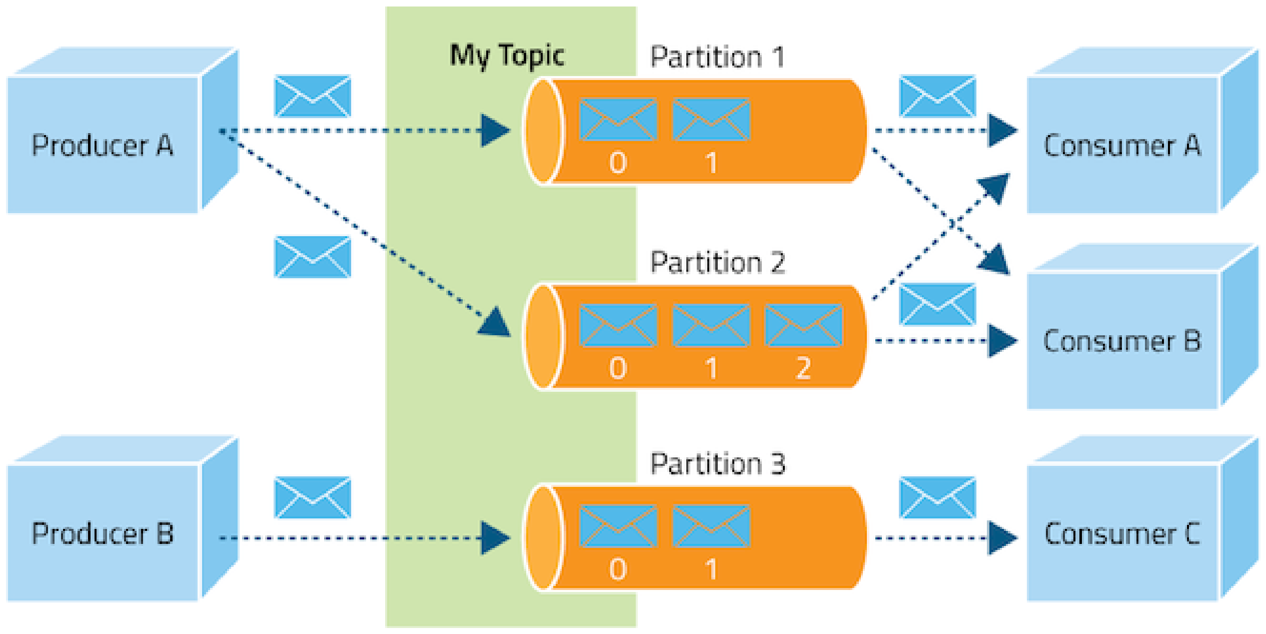
- **Built-in Stream Processing**: Kafka supports processing streams of events with features like joins, aggregations, filters, transformations, and more, utilizing event-time and exactly-once processing. This enables real-time data processing and analytics directly within the Kafka platform, eliminating the need for external processing frameworks.
- **Connectivity**: With Kafka's Connect interface, users can seamlessly integrate with hundreds of event sources and sinks, including databases like Postgres, messaging systems like JMS, data stores like Elasticsearch, and cloud services like AWS S3. This simplifies data integration and enables Kafka to act as a central hub for streaming data.
- **Client Libraries**: Kafka offers client libraries in various programming languages, enabling users to read, write, and process streams of events. This allows developers to interact with Kafka using their preferred programming language, making it easy to integrate Kafka into existing applications and workflows.
- **Large Ecosystem of Open Source Tools**: Leveraging a vast array of community-driven tooling, Kafka users benefit from a rich ecosystem of open-source tools. These tools extend Kafka's functionality, providing additional capabilities for monitoring, management, data processing, and more.
- **Reliability and Ease of Use**: Kafka is suitable for mission-critical applications, offering guaranteed ordering, zero message loss, and efficient exactly-once processing. Trusted by thousands of organizations, Kafka has amassed over 5 million unique lifetime downloads, highlighting its reliability and ease of use.
- **Vast User Community**: Kafka boasts a vibrant user community, actively contributing to its development and support. It is one of the five most active projects of the Apache Software Foundation, with numerous meetups worldwide and rich online resources, including documentation, training, tutorials, videos, and community forums.

With these key features, Kafka continues to revolutionize event streaming, making it the preferred choice for companies across various industries. Now, let's analyze some basic terms used in Kafka:

- **Topic**: In Kafka, a topic is a user-defined category or feed name where data is stored and published. It serves as a logical channel for organizing and categorizing streams of records.
- **Producer**: A producer is an application or component responsible for publishing data to Kafka topics. Producers generate events or records and send them to one or more Kafka topics for storage and distribution.
- **Consumer**: A consumer is an application or component responsible for reading data from Kafka topics. Consumers subscribe to one or more topics and consume events or records published to those topics by producers.
- **Consumer Group**: A consumer group is a logical grouping of consumers that jointly consume data from one or more Kafka topics. Each consumer within the group reads from a subset of the partitions of the subscribed topics. Note that each message can be consumed only from one consumer per consumer group.

- **Broker**: A broker is a Kafka server responsible for handling requests from producers and consumers. Brokers manage the storage and replication of topic partitions, handle client connections, and coordinate data replication and distribution across the Kafka cluster.
- **Partition**: A partition is a unit of parallelism and distribution within a Kafka topic. Topics can be divided into multiple partitions, each serving as an ordered and immutable sequence of records.

A visualization of the Kafka infrastructure is presented below:



9. Kafka Architecture [46]

4.1.3 Spring Framework

The Spring Framework [43] is a comprehensive and modular framework for building enterprise applications in Java. It provides a wide range of features and functionalities to simplify the development process and enhance the scalability, maintainability, and testability of applications. Some key features of the Spring Framework include:

- **Inversion of Control (IoC) Container**: The core of the Spring Framework is its IoC container, which manages the lifecycle of Java objects (beans [27]) and their dependencies. With IoC, the framework takes control of creating, configuring, and managing objects, reducing the complexity of application development.
- **Aspect-Oriented Programming (AOP)**: Spring supports AOP, allowing developers to modularize cross-cutting concerns such as logging, transaction management, security, and caching. AOP enables the separation of concerns, leading to cleaner and more maintainable code.

- **Dependency Injection (DI)**: DI is a core principle of the Spring Framework that enables loose coupling between components. By injecting dependencies into beans rather than hard-coding them, DI promotes reusability, testability, and flexibility in application development.
- **Spring MVC**: Spring MVC is a web framework built on top of the Spring Framework, providing robust support for building web applications. It follows the Model-View-Controller (MVC) architecture, allowing developers to separate concerns and build scalable and maintainable web applications.
- **Integration with other frameworks and technologies**: Spring integrates seamlessly with various Java technologies and third-party frameworks, including Hibernate, JPA, JDBC, JMS, and more. It provides comprehensive support for enterprise features such as transaction management, security, messaging, and scheduling.

4.1.4 Spring Boot

Spring Boot [44] is an opinionated framework built on top of the Spring Framework, designed to simplify the process of building and deploying production-ready Spring applications. It provides a convention-over-configuration approach, eliminating the need for boilerplate code and configuration. Some key features of Spring Boot include:

- **Auto-configuration**: Spring Boot automatically configures the application based on its dependencies and the environment. It eliminates the need for manual configuration by providing sensible defaults, allowing developers to focus on writing business logic rather than setting up infrastructure.
- **Standalone executables**: Spring Boot applications can be packaged as standalone executables (JAR or WAR files) with an embedded servlet container (such as Tomcat, Jetty, or Undertow). This simplifies deployment and eliminates the need for external application servers.
- **Spring Boot starters**: Spring Boot starters are a set of pre-configured dependencies that streamline the development of specific types of applications, such as web applications, data-driven applications, messaging applications, and more. Starters encapsulate common configurations and dependencies, making it easy to get started with Spring Boot projects.
- **Actuator**: Spring Boot Actuator provides built-in monitoring and management capabilities for Spring Boot applications. It exposes various endpoints for monitoring application health, metrics, environment properties, and more. Actuator enables developers to monitor and manage applications in production without writing additional code.

4.1.5 Java Persistence API (JPA)

Java Persistence API (JPA) [45] is a Java specification for managing relational data in Java applications. It provides a set of standard interfaces and annotations for mapping Java objects to relational database tables and executing database operations. Some key features of JPA include:

- **Entity mapping**: JPA allows developers to map Java classes (entities) to database tables and define relationships between entities using annotations such as @Entity, @Table, @JoinColumn, and @OneToMany.
- **Object-relational mapping (ORM)**: JPA supports ORM frameworks such as Hibernate, EclipseLink, and OpenJPA, which handle the translation between Java objects and relational database tables. ORM frameworks abstract away the low-level JDBC code, making it easier to work with databases in Java applications.
- **Entity lifecycle management**: JPA provides lifecycle callbacks such as @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, and @PostRemove, allowing developers to hook into entity lifecycle events and execute custom logic.
- **JPQL (Java Persistence Query Language)**: JPA includes JPQL, a query language similar to SQL but tailored for working with Java objects. JPQL allows developers to write database queries using entity classes and properties, rather than SQL queries.

With the usage of the three frameworks we described before, it's straightforward to create various applications in Java. The main components of those applications will always contain all or some of the following objects:

- **Controllers**: In Spring MVC and Spring Boot applications, controllers are responsible for handling incoming HTTP requests, processing user input, and returning appropriate responses. Controllers typically contain request mappings (annotated with @RequestMapping or similar annotations) that map URLs to handler methods.
- **Services**: Services in Spring applications encapsulate business logic and perform operations such as data validation, transformation, and manipulation. Services interact with repositories to retrieve and store data and are often used to coordinate transactions and manage application state.
- **Entities**: In JPA, entities are plain Java objects (POJOs) that represent persistent data stored in a relational database. Entities are typically annotated with JPA annotations such as @Entity, @Table, and @Column to define their mapping to database tables and columns.
- **Repositories**: Repositories in Spring applications provide a higher-level abstraction for performing database operations on entities. They typically extend JpaRepository or a similar interface provided by Spring Data JPA and provide methods for common CRUD (Create, Read, Update, Delete) operations as well as custom queries.

Additionally, those frameworks make it easy to change the underlying database by providing abstraction layers and standard APIs. By using JPA for database access, developers can write database-independent code that is not tied to a specific database vendor. Spring Data JPA further simplifies database access by providing a repository abstraction that decouples the application from the underlying data access technology. Also, Spring Boot's autoconfiguration and dependency management features streamline the process of configuring data sources and integrating with different databases. Developers can switch between relational databases (such as MySQL, PostgreSQL, Oracle, or SQL Server) by simply changing the database driver and connection properties in the application configuration.

Overall, the combination of Spring Framework, Spring Boot, and JPA provides a flexible and scalable architecture for building Java-based applications that can easily adapt to changing database requirements and environments.

4.1.6 Peer-To-Peer (P2P)

Peer-to-peer (P2P) architecture [33, 36, 37, 38] decentralizes networking, enabling each node to serve as both a client and a server, thereby eliminating the need for central coordination. In a P2P network, every node has equal status, and there is no central authority controlling the flow of information. This architecture contrasts with the traditional client-server model, where clients request services from central servers that provide resources. In a P2P network, each node can initiate requests and respond to them, allowing for distributed sharing of resources and workload. One of the key advantages of P2P architecture is its resilience against single points of failure. Since there is no central server, the network can continue to function even if some nodes fail or leave the network. Additionally, P2P networks can scale more efficiently, as the addition of new nodes increases the overall resources available to the network.

4.1.7 Distributed Hash Table (DHT)

Distributed hash tables (DHTs) [32, 34, 35] provide a scalable and fault-tolerant mechanism for locating and retrieving data within a decentralized network. A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table. It partitions a key space across a decentralized network and uses hash functions to assign ownership and locate values. DHTs enable efficient routing of key-value queries within the network, ensuring redundancy and load distribution. In a DHT, each node in the network is responsible for storing and managing a portion of the key space. When a client wants to retrieve or store data, it sends a request to the network, which is then routed to the appropriate node based on the key of the data. The node responds to the request and either retrieves or stores the data accordingly. DHTs coordinate between untrusted distributed nodes to route key-value queries securely and efficiently using overlay networks. Nodes locally store values for a subset of keys and collaborate to correctly route queries. They provide a scalable way to distribute load and redundancy across machines and offer guaranteed lookup times within the network diameter.

4.2 System Architecture and Technology Stack

To achieve a system that stores data at the edge with a high level of security, a novel architecture has been developed utilizing SQLite technology. In this innovative design, each node hosts its dedicated SQLite instance, meticulously preserving pertinent data generated by the respective node.

A crucial aspect of this implementation is the adoption of industry-standard technologies, ensuring robustness and compatibility across diverse organizational ecosystems. The system uses Java 21 (OpenJDK 21.0.2), the latest LTS Java version, known for its reliability, stability, and speed improvements over previous versions. Spring 3.1, coupled with Hibernate 6, introduces a suite of optimizations aimed at improving performance, security, and scalability. Noteworthy advancements in Hibernate 6 empower the system with enhanced data management capabilities, bolstering efficiency and fortifying data integrity.

Apache Maven 3.9.6 facilitated project management by managing dependencies and efficiently building the project. Docker version 24.0.7 served as the containerization platform, providing a consistent and reproducible environment.

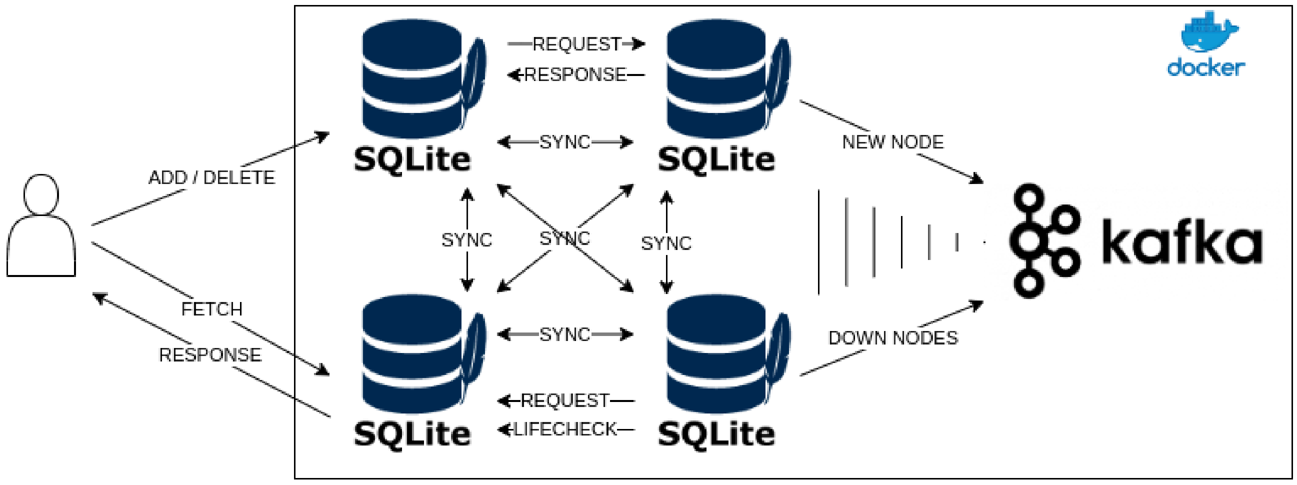
Furthermore, the incorporation of Kafka enhances the system's functionality, facilitating seamless communication and action execution across distributed nodes. By leveraging Kafka's event-driven architecture, the system achieves decoupled communication, enabling nodes to execute tasks autonomously without extensive coordination with other network entities.

Project Lombok is employed to streamline development and enhance code maintainability, effectively reducing boilerplate code in Java classes. Lombok automates common tasks such as generating getters, setters, and constructors, thereby improving code readability and accelerating development cycles.

These software versions were meticulously selected for their reliability and compatibility, ensuring smooth development and replication of the thesis. This architecture not only mitigates the challenges posed in the use cases of section 1.1 but also sets a precedent for scalable and resilient systems architecture in enterprise environments. It serves as a testament to the convergence of cutting-edge technologies and optimal practices in the pursuit of innovation and operational excellence.

4.3 System explanation

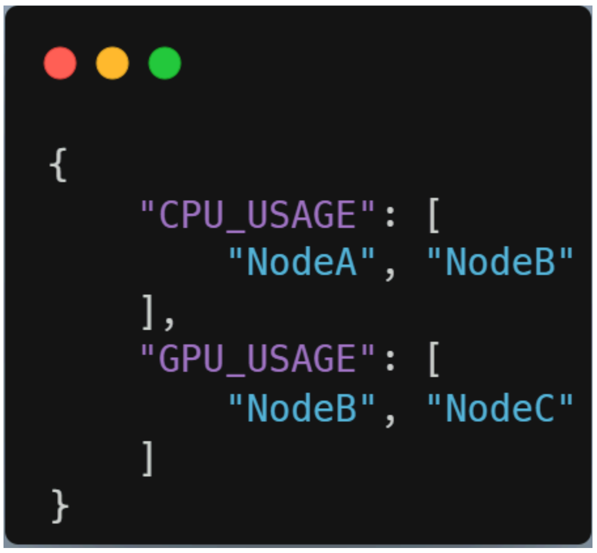
Before analyzing the main classes/components of our application separately, let's proceed with an analysis of the flows which are used by our application. Below, you can see a diagram illustrating the various actions executed within our system and the components that interact for each action:



10. Architecture of the implemented framework

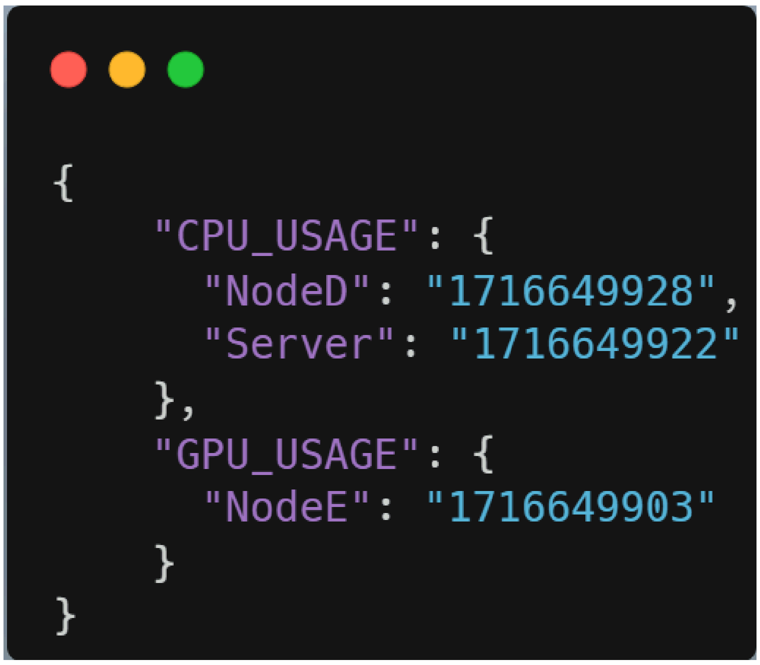
Our application starts when the Kafka service is launched in the Docker network, and one or more SQLite nodes enter the Docker network. Each SQLite node, in addition to the embedded SQLite database, also contains a set of various functionalities and services to handle the different flows of the framework. At this stage, we need to take a first look at the DhtService, which runs inside the SQLite node. This service is responsible for communication between every node in the network. It contains four crucial objects:

1. **existingNodes**: This object is a set of strings (Set<String>) containing the nodes known to the current node. This field is used during synchronization and specific fetch requests.
2. **dataMap**: This object is a hashMap of metric types to a set of strings representing the nodes that contain metrics of this type. It is used during fetch requests to avoid unnecessary requests. An example of this object is visible in the following image:


```
{
    "CPU_USAGE": [
        "NodeA", "NodeB"
    ],
    "GPU_USAGE": [
        "NodeB", "NodeC"
    ]
}
```

11. Example of “dataMap” object

3. **deletedTypes**: This object is a set of metric types (Set<EntityTypeE>) used during synchronization to inform other nodes that they no longer contain metrics of the specified metric types.
4. **deletedTypesMap**: This object is a map from metric type to another map, where the key is a string indicating the node name, and the value is a timestamp indicating when this metric type was deleted from the given node. It ensures that the system remains unaffected by synchronization requests with invalid information. An example of this object is visible in the following image:



```
{
    "CPU_USAGE": {
      "NodeD": "1716649928",
      "Server": "1716649922"
    },
    "GPU_USAGE": {
      "NodeE": "1716649903"
    }
}
```

12. Example of “deletedTypesMap” object

Further details about the DhtService can be found in Section 4.5.6.2. When a node is added to the network, it initiates the "New Node" flow, which will be discussed in Section 4.3.1. Additionally, every five minutes, each node executes the "Sync" flow, described in Section 4.3.2, to share its internal abstract information (existingNodes, dataMap and deletedTypes) with the other nodes. Each online node provides users with the following capabilities:

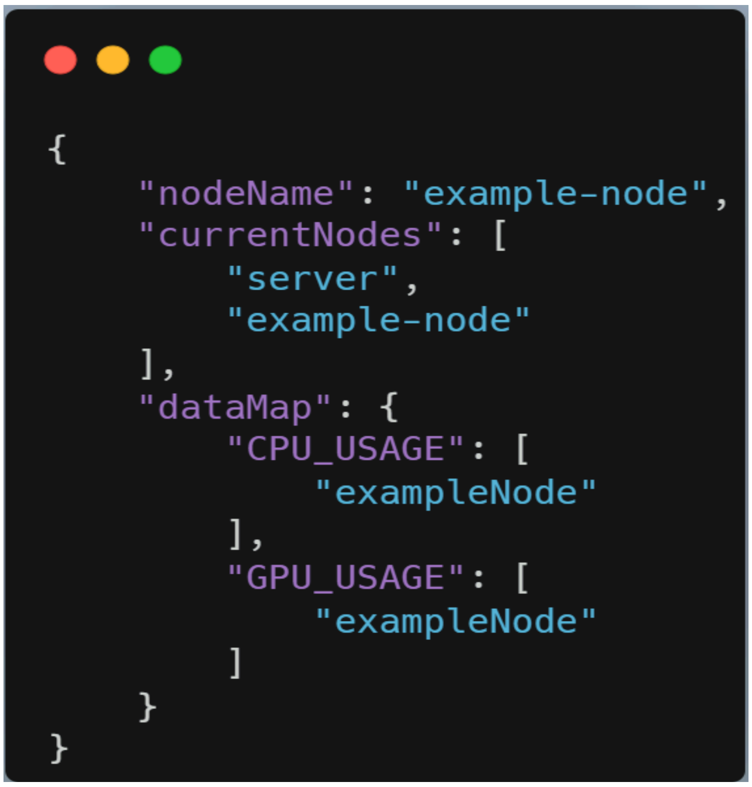
- **Add**: Users can add a metric, container, or pod to the database of the target node. This request triggers the "Add" flow, detailed in Section 4.3.3.
- **Delete**: Users can remove a metric from the database of the target node. This request triggers the "Delete" flow, discussed in Section 4.3.4.
- **Fetch**: Users can retrieve metrics from the node they query, from a specified node, or from all nodes in the network (if a specific metric type is indicated). This request triggers the "Fetch" flow, analyzed in Section 4.3.5. In certain cases, as a result of the “Fetch” flow, "Lifecheck" and "Down Nodes" flows can be triggered, which are detailed in Sections 4.3.6 and 4.3.7, respectively.

Having provided an overview of the framework, we will now proceed to a detailed analysis of the flows executed within the system.

4.3.1 New Node Flow

When a new node is started then there are various actions which take place in the new node. This process is visible in figure 14 and the steps are described below:

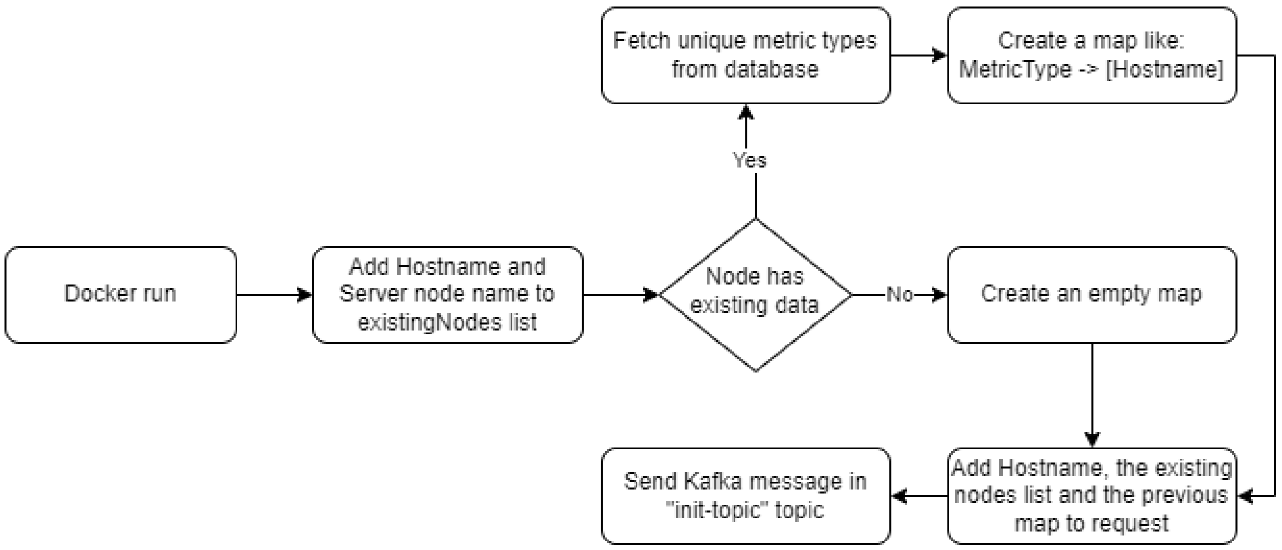
1. To add itself to the network, the new node adds itself and the server node to the "existingNodes" set of DhtService
2. If this node has an existing database, which means that the node was up in the past, then the node was to initialize the "dataMap" object. To do so, the unique metric types are fetched from the database e.g. "MEMORY_USAGE, CPU_USAGE". Then we loop through this list and for each element we add a new entry to "dataMap" in order to connect the metric type with the hostname of the new node e.g. "MEMORY_USAGE -> ['NewNode'] ". If there isn't an existing database, the "dataMap" is initialized with an empty map.
3. In this step, the node has initialized every object with the internal information. As a result, the node is ready to inform the rest nodes that it is online. To do so, it creates an object with the necessary details in order to share the metric types which are included in it as well as its name. An example of the object which is created is visible below:



```
{
    "nodeName": "example-node",
    "currentNodes": [
        "server",
        "example-node"
    ],
    "dataMap": {
        "CPU_USAGE": [
            "exampleNode"
        ],
        "GPU_USAGE": [
            "exampleNode"
        ]
    }
}
```

13. Example of "init-topic" object

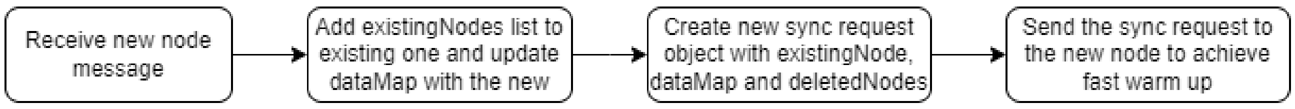
4. The final step is to send this object through kafka. The message will be sent in the “init-topic” topic. This message will be handled only from one node.


Fetch unique metric types from database
Create a map like: MetricType -> [Hostname]
Yes
Docker run
Add Hostname and Server node name to existingNodes list
Node has existing data
No
Create an empty map
Send Kafka message in "init-topic" topic
Add Hostname, the existing nodes list and the previous map to request

14. “Init Node” flow diagram

As previously mentioned, the Kafka message sent from the new node to the “init-topic” topic is handled by a single node. The node that processes this message is responsible for two main tasks: a) assisting the new node to achieve a quick warm-up so it can respond to requests, and b) updating its own objects with the information from the new node to inform the other nodes during its “Sync” flow. The actions taken by the node handling the request are illustrated in the image 15 and described below:

1. The node's first task is to update the “existingNodes” and “dataMap” objects with the abstract information from the received message. Note that the “deletedTypes” is not updated since it cannot be populated by a node that has just started.
2. Once the internal objects have been updated, a new copy of these objects is created and a sync request is prepared. This request includes the “existingNodes”, “dataMap”, and “deletedTypes” of the current node.
3. This sync request is then sent to the new node. The handling of this request by the new node will be analyzed in Section 4.3.2.

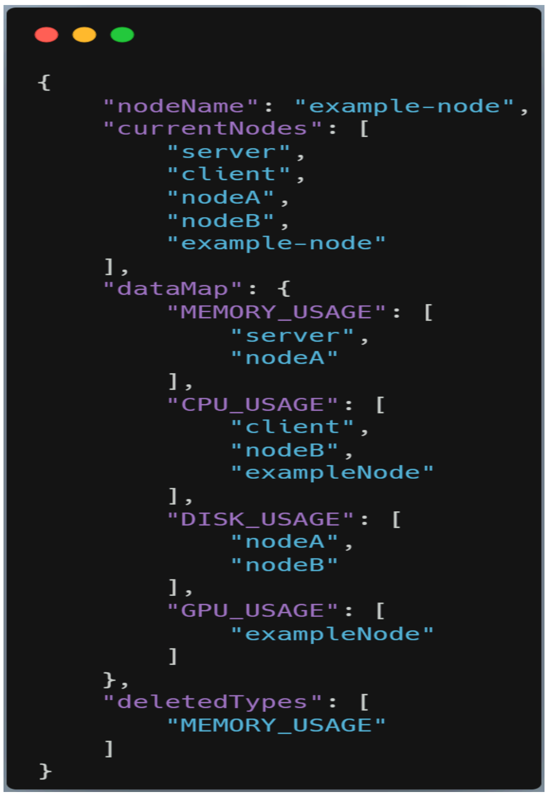

Receive new node message
Add existingNodes list to existing one and update dataMap with the new
Create new sync request object with existingNode, dataMap and deletedNodes
Send the sync request to the new node to achieve fast warm up

15. Actions after receival of “init-topic” message

4.3.2 Sync Flow

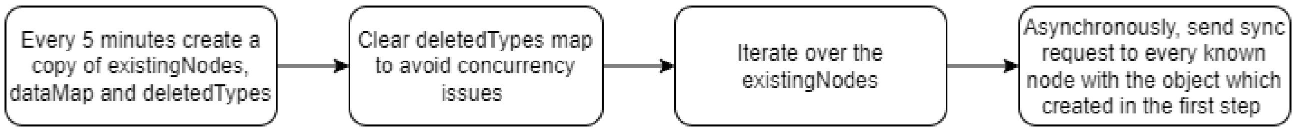
Every five minutes, each node attempts to send its data to every other node in the network to inform them about changes in its content. Specifically, the shared content includes the dataMap, existingNodes, and deletedTypes. The syncing process of the node that sends the request is visualized in figure 17 and described in detail below:

1. When five minutes have passed since the last sync, the node initiates the syncing process. It creates a copy of the objects that need to be sent to the other nodes to ensure minimal lock time, as the system cannot respond to any other requests during the lock.
2. To avoid resending the deletion of a metric type from this node, the deletedTypes object is cleared after creating the copy of the aforementioned objects.
3. The process then iterates through the existingNodes set to send the sync request to every known node. As an enhancement, this step can be replaced with a Kafka message that will be handled by every other node.
4. Finally, for each node in the existingNodes set, the sync object created in the first step is sent. The request is sent asynchronously to ensure that the execution time of the sync action is not affected by the state and response time of the other nodes. An example of the sync object is shown below:



```
{
    "nodeName": "example-node",
    "currentNodes": [
        "server",
        "client",
        "nodeA",
        "nodeB",
        "example-node"
    ],
    "dataMap": {
        "MEMORY_USAGE": [
            "server",
            "nodeA"
        ],
        "CPU_USAGE": [
            "client",
            "nodeB",
            "exampleNode"
        ],
        "DISK_USAGE": [
            "nodeA",
            "nodeB"
        ],
        "GPU_USAGE": [
            "exampleNode"
        ]
    },
    "deletedTypes": [
        "MEMORY_USAGE"
    ]
}
```

16. Example of “sync” object

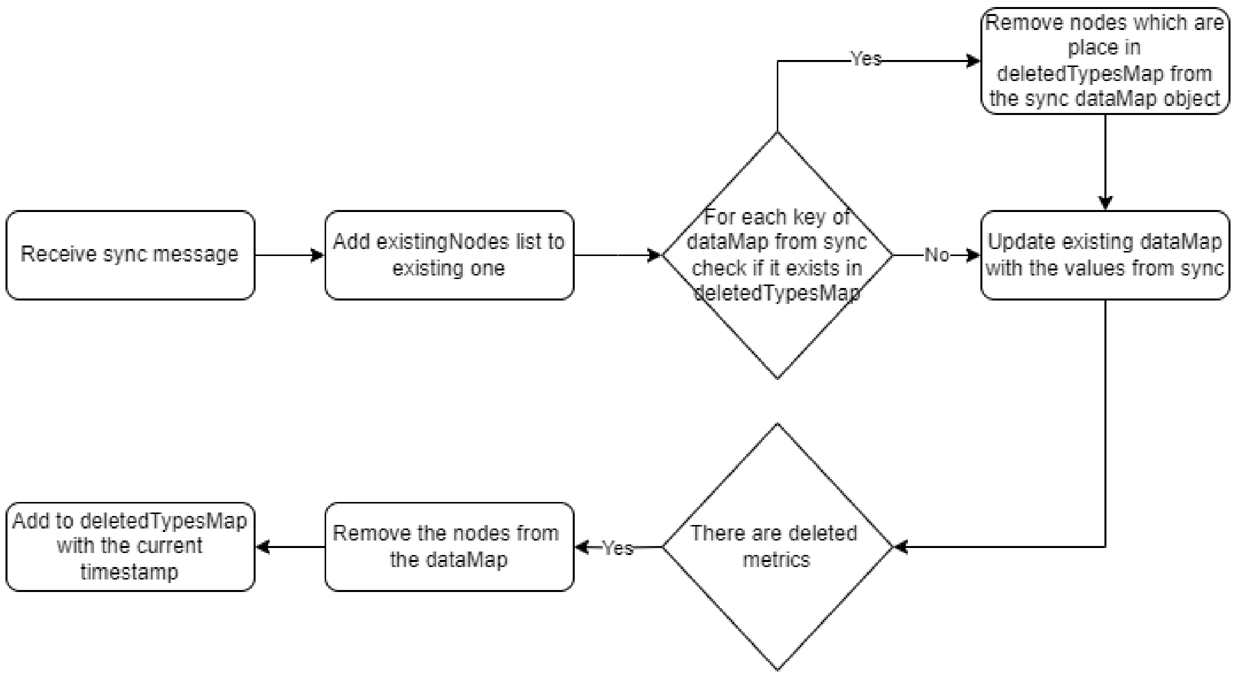

Every 5 minutes create a copy of existingNodes, dataMap and deletedTypes
Clear deletedTypes map to avoid concurrency issues
Iterate over the existingNodes
Asynchronously, send sync request to every known node with the object which created in the first step

17. Generation of sync request

When a node receives a sync request from another node, either during the initialization of a new node or during the regular sync flow, it must follow specific steps. The flow is illustrated in image 18, and the steps are detailed below:

1. The first action is to update the existingNodes set with the one included in the sync object. This allows the node to recognize and incorporate new nodes into its system.
2. Next, the node iterates through the dataMap object from the sync request. For each key, it checks if the key is included in deletedTypes. If so, the node associated with that key is removed from the dataMap to avoid inserting invalid information. If a node had previously deleted a metric type but now includes it again, the information will be updated in the next sync object, as the deprecation period will have passed. Once the dataMap entries of the sync object are cleaned up, the node updates its own dataMap.
3. The final step is to process the deletedTypes object from the sync request. If this object is empty, no further action is needed. Otherwise, the node checks each entry in the deletedTypes object and updates its dataMap to remove the corresponding nodes. If a key in the dataMap becomes empty after this update, the key is also removed. Finally, the node updates its deletedTypes map by adding the entries from the deletedTypes object of the sync request, each with the current timestamp. This allows for the removal of entries after 10 minutes, enabling the re-insertion of previously deleted metric types.

By following these steps, the node ensures that its internal state is consistent and up-to-date with the latest changes in the network.

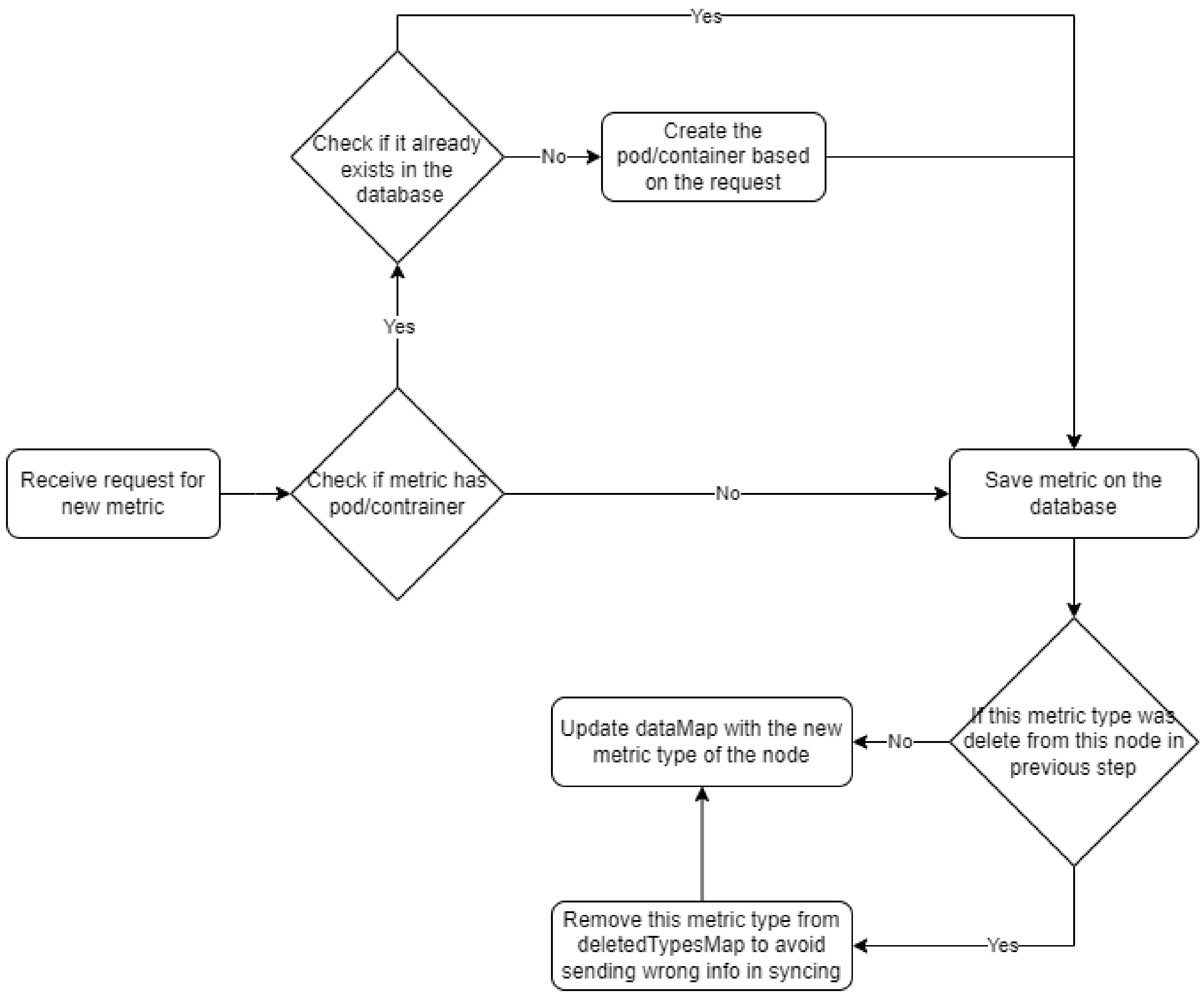

Receive sync message
Add existingNodes list to existing one
For each key of dataMap from sync check if it exists in deletedTypesMap
Yes
Remove nodes which are place in deletedTypesMap from the sync dataMap object
No
Update existing dataMap with the values from sync
There are deleted metrics
Yes
Remove the nodes from the dataMap
Add to deletedTypesMap with the current timestamp

18. Diagram of actions after the retrieval of a sync request

4.3.3 Add Flow

The users are able to insert data to each node. In order to do so, they have to send a request to the node to start the "Add" flow. A diagram of the "Add" flow execution is visible in figure 19 and the required steps are described below:

1. When the system receives an add request for a metric, it has to check if the metric is related to a pod or container. Note that the user is able to add a pod or a container with a similar request which is not analyzed extensively in the scope of this section. If the new metric is related to a pod or a container then the system checks if this pod/container already exists. If not then it proceed with the insertion of the pod/container to the database
2. Once this is done, the system adds the new metric to the database of the node.
3. The final step is to check if the metric type of the metric which was just added to the database was present in the deletedTypes map. If so, the system removes this metric from the deletedTypes map to ensure that the node won't send invalid information during syncing. Next, it updates the dataMap object in order to contain the correct abstract information.


Yes
Check if it already exists in the database
No
Create the pod/container based on the request
Yes
Receive request for new metric
Check if metric has pod/contrainer
No
Save metric on the database
If this metric type was delete from this node in previous step
No
Update dataMap with the new metric type of the node
Yes
Remove this metric type from deletedTypesMap to avoid sending wrong info in syncing

19. Diagram of "Add" flow

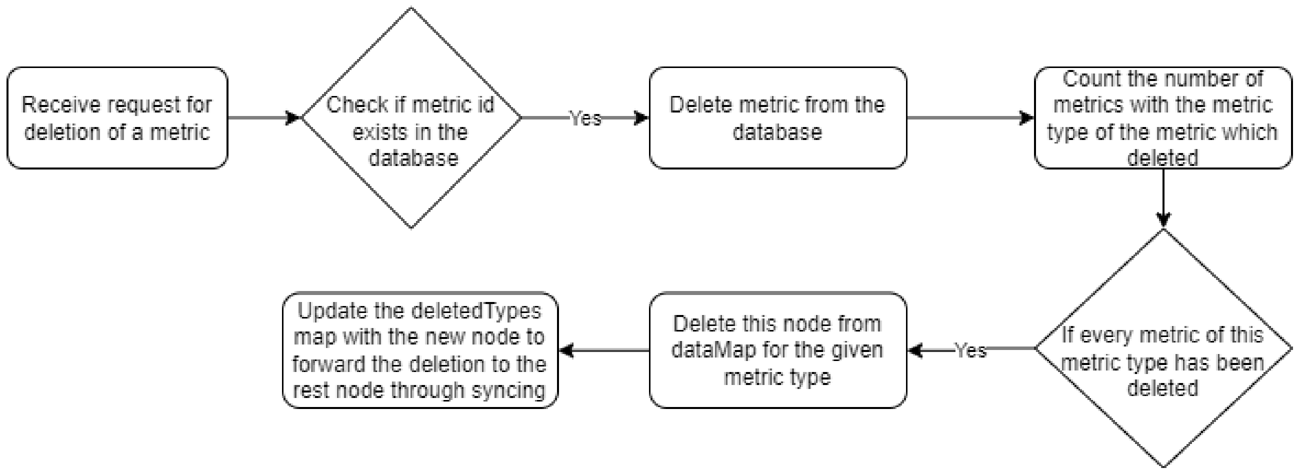
4.3.4 Delete Flow

Having completed the analysis of the "Add" flow, let's proceed to the analysis of the "Delete" flow. Any user can delete a specific metric from the database of a node by sending a request with the ID of the target metric. The steps that make up the "Delete" flow are illustrated in image 20 and described below:

1. First the system checks if the candidate metric for deletion actually exists in the database. If it does not exist then a warning message is printed and the flow completes its execution.
2. After ensuring that the metric exists, the system proceeds to the actual deletion of the metric.
3. To ensure consistency in the system and to avoid unnecessary requests, the system checks if the metric type of the deleted metric is still present in the database. To do so, the system

fetch the count of the entries with the specific metric type e.g. MEMORY_USAGE from the database.

4. If the return value is greater than zero (> 0), then the deletion of the metric is done. In different case, the system has to update the deletedTypes, deletedTypesMap and the dataMap objects. First it removes the current node from the list of the specific metric type from the dataMap object. Then it adds the metric type of the delete metric to the deletedTypes set in order to inform the rest nodes of the network that the specific metric type has been removed from the current node. Finally, it enters a new entry to the deletedTypesMap with the current timestamp in order to ensure that the node won't receive invalid information from another node.


Receive request for deletion of a metric
Check if metric id exists in the database
Yes
Delete metric from the database
Count the number of metrics with the metric type of the metric which deleted
If every metric of this metric type has been deleted
Yes
Delete this node from dataMap for the given metric type
Update the deletedTypes map with the new node to forward the deletion to the rest node through syncing

20. Diagram of "Delete" flow

4.3.5 Fetch Flow

One of the most significant flows in our framework is the "Fetch" flow. Through this flow, the user can fetch data from the target node, another specified node, or every node in the network based on the specified metric type. The diagram of the "Fetch" flow is displayed in figure 22. The green arrows symbolize the requests occurring inside the node, and the caller waits for the response from the target node. To achieve a better level of understanding, let's analyze this flow step by step:

When the system receives a new fetch request, it has to determine the process to follow. The first check involves understanding the scope of the request. The system checks if the request specifies a node or not.

- If the request does not specify a node:

    a. The system checks if there is a specified metric type. If neither a node nor a metric type is specified, the system fetches every metric from the database of the node that received the request and returns it to the user.
    b. If a metric type is specified, the system requests the entries of the specified metric type from every node in the network. To avoid overloading the network with unnecessary calls, the system uses the dataMap object to determine which nodes actually have metrics of the specified type.
    c. If the list of nodes, as determined by dataMap, is empty, the system returns an empty list as the result. This can happen because no node has metrics of the specified type or because the nodes that have such metrics have not performed their sync request yet, so the current node is not aware of those metrics.
    d. If there are nodes in the final list, the system sends a fetch request for the specified metric type from the database of the node that received the request to every node in this list. To understand the flow that will be followed in the inner request, we can go back to step 1 and re-run the steps based on the new arguments. The system will wait 4 seconds to receive results from every node. Note that this value is the default one but it should be configured based on the use case. For example, if the returned data are expected to be more than 50 Mb it should be configured to something bigger as it is normal to take longer for that amount of data.
    e. If some nodes throw an exception during the request, the system will execute the "Lifecheck" flow for those nodes.
    f. For the nodes that respond within this time window, the system will add their metrics to a HashMap, with the name of the source node as the key and a list of metrics from this node as the value. Metrics from nodes that fail to respond within 4 seconds will be ignored in the final response. This is to overcome performance and network issues that may appear randomly. As a result, the system will never take more than 4 seconds to respond.

- If the request specifies a node:
    a. If the target node is the current node:
        i. If the request specifies a metric type, the system requests every metric of the specified type from the database of the current node and returns them to the caller.
        ii. If the request does not specify a metric type, the system requests every metric from the database of the current node and returns them to the caller.
    b. If the target node is not the current node:
        i. First, we need to check if the current node is aware of the specified node to skip unnecessary steps if the target node is unknown to the current node or does not exist in the network. The system searches in the existingNodes set. If there is no match for the target node, the system returns an empty list to the caller.
        ii. If the target node is contained in the existingNodes set, the system checks if the request specifies a metric type. If so, the system checks the dataMap

object to determine whether the specified node is included in the list for the specified metric type. If not, the system returns an empty list to the caller as there are no matches for the specified arguments. Otherwise, the flow proceeds to the next step.

iii. Having ensured that the target node exists and either there is no specified metric type or the specified metric type exists in the target node, the system sends an inner request to the target node with the necessary arguments to fetch the appropriate data. In this case, the system waits for the response from the target node. If an exception occurs during the request, an empty list is returned to the caller. To understand the flow that will be followed in the inner request, we can go back to step 1 and re-run the steps based on the new arguments.

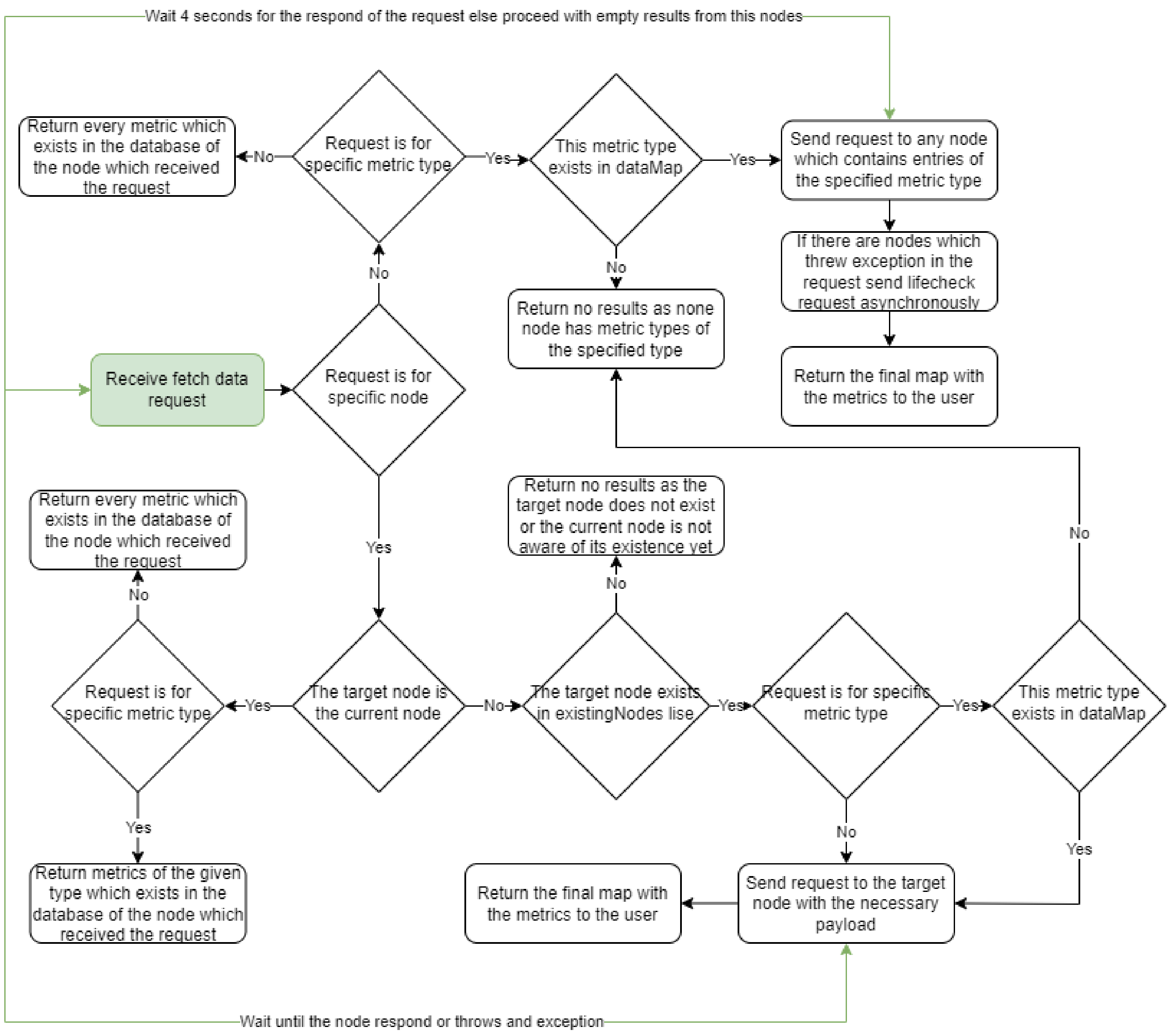
iv. Finally, when the current node receives the response, it sends the response of the inner call to its caller.

As this flow is the most complex flow in our framework, let's go through an execution example to achieve better understanding. Let's assume that we have a node named “NodeB” with existingNodes = ["NodeA", "NodeB"] and dataMap = { "MEMORY_USAGE": ["NodeA"], "CPU_USAGE": ["NodeA", "NodeB"] }. A new fetch request is received from NodeB with targetNode="NodeA" and metricType="MEMORY_USAGE". The steps which will be followed are:

1. The system checks for the target node, and the target node is "NodeA".
2. The system checks if this node is the current node and determines that it is not the current node since the request is received from NodeB.
3. The system checks in the existingNodes set and finds NodeA in it, so it proceeds to the next step.
4. The system checks if the request is for a specific metric type, and based on the request, it determines that the request is for the MEMORY_USAGE metric type.
5. It checks the dataMap for the MEMORY_USAGE key, which has a value of ["NodeA"]. The target node is included in this list, so the system proceeds to the inner request. The inner request will be sent to NodeA and will have the same arguments (targetNode="NodeA" and metricType="MEMORY_USAGE").
    a. The system checks for the target node, and the target node is "NodeA."
    b. The system checks if this node is the current node and finds that the target node is the current node.
    c. The system checks if there is a specified metric type and finds that the specified metric type is MEMORY_USAGE.
    d. It fetches the metrics with metricType = MEMORY_USAGE from the database of the current node and returns those metrics to the caller. In this case, the caller is NodeB.

6. NodeB retrieves the response from NodeA and returns those metrics to its caller, which in this case is the user who sent the initial request.

This example demonstrates how the system processes a fetch request involving a target node and a specific metric type, ensuring accurate data retrieval and handling each step methodically.



Wait 4 seconds for the respond of the request else proceed with empty results from this nodes
Return every metric which exists in the database of the node which received the request
No
Request is for specific metric type
Yes
This metric type exists in dataMap
Yes
Send request to any node which contains entries of the specified metric type
If there are nodes which threw exception in the request send lifecheck request asynchronously
Return the final map with the metrics to the user
No
No
Return no results as none node has metric types of the specified type
Receive fetch data request
Request is for specific node
Return every metric which exists in the database of the node which received the request
Return no results as the target node does not exist or the current node is not aware of its existence yet
No
Yes
No
No
Request is for specific metric type
Yes
The target node is the current node
No
The target node exists in existingNodes lise
Yes
Request is for specific metric type
Yes
This metric type exists in dataMap
Yes
No
Yes
Return metrics of the given type which exists in the database of the node which received the request
Return the final map with the metrics to the user
Send request to the target node with the necessary payload
Wait until the node respond or throws and exception

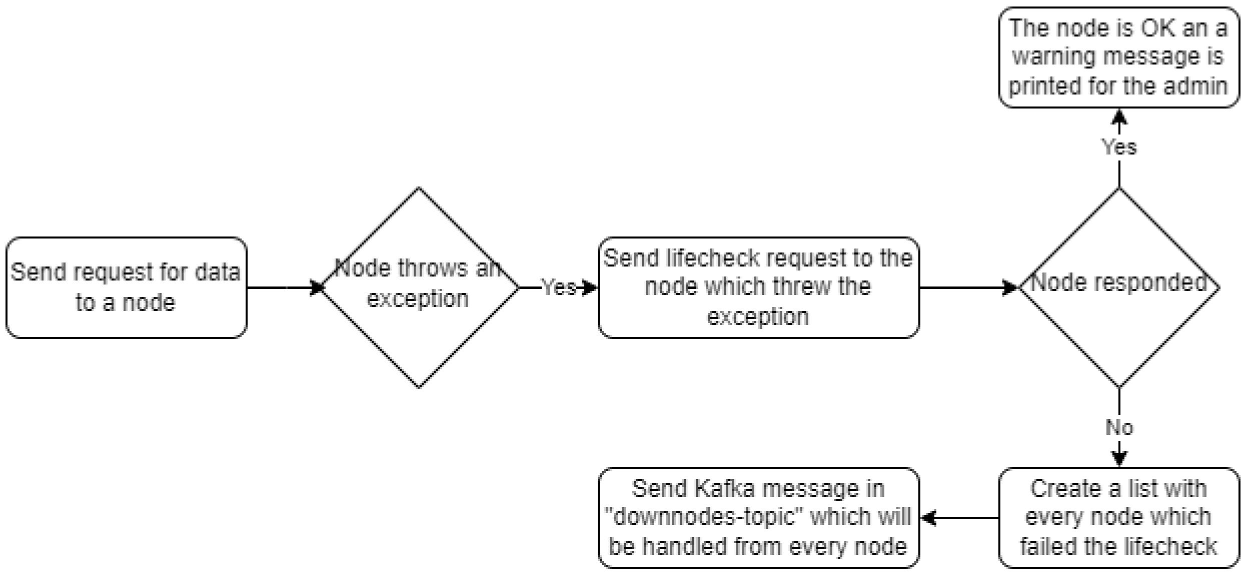
21. Diagram of “Fetch” flow

4.3.6 Lifecheck Flow

When a node encounters an exception while fetching its data, and the request originated from another node within the network, the "Lifecheck" flow is executed. This flow determines whether the node that threw the exception is actually down or if the issue was due to a temporary networking problem. The steps of the "Lifecheck" flow are visualized in image 23 and described as follows:

1. If an exception occurs during data fetching from a node, the system initiates the lifecheck process.
2. The system compiles a list of all nodes that throw exceptions and sends a simple GET request to each one. Nodes that respond are marked as OK and are excluded from further steps. Nodes that fail to respond are marked as DOWN and proceed to the next step.
3. After collecting responses, the system creates a list of nodes that failed to respond.
4. Finally, the system sends a new Kafka message to the “downnodes-topic” topic containing the list of unresponsive nodes. This message is processed by every node in the network. When a node receives this message, it starts the "Down Nodes" flow, detailed in Section 4.3.7.

This process ensures that the network accurately identifies and handles nodes that are down, maintaining the overall health and stability of the system.


The node is OK an a warning message is printed for the admin
Yes
Send request for data to a node
Node throws an exception
Yes
Send lifecheck request to the node which threw the exception
Node responded
No
Send Kafka message in "downnodes-topic" which will be handled from every node
Create a list with every node which failed the lifecheck

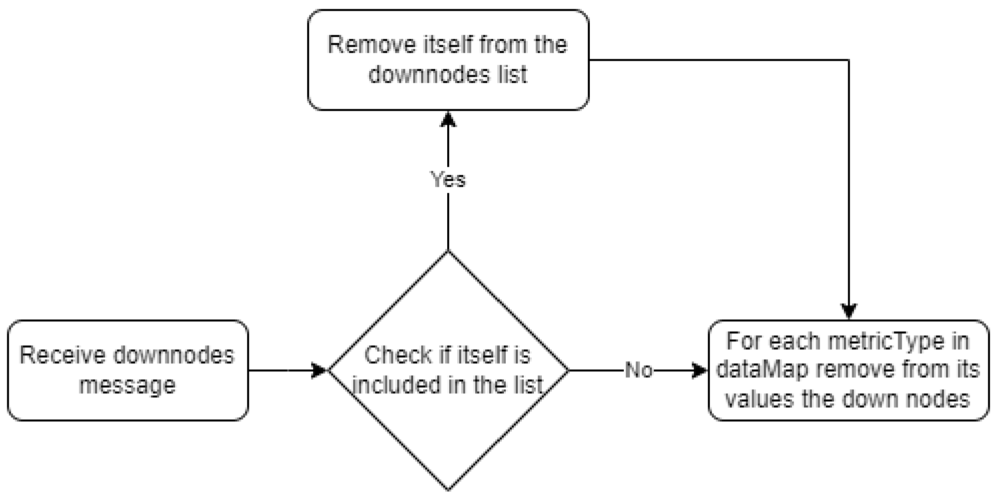
22. Diagram of “Lifecheck” flow

4.3.7 Down Nodes Flow

The “Down Nodes” flow is executed when a node receives a Kafka message on the “downnodes-topic”. This flow is designed to remove any down nodes from the system to avoid delays in future requests. If the node that threw the exception is still online, it will be re-inserted into the network when it sends its next sync request. If the node is down, it will be re-inserted into the network only if it is restarted and executes its “Init Node” flow. The visualization of the “Down Nodes” flow is provided in figure 24, and the steps are detailed below:

1. A node receives a message on the “downnodes-topic”. This message contains a list of nodes that were down during the last fetch request.
2. Before taking any action, the system checks if the current node is included in this list. If it is, the current node is removed from the list, as it is still online (evidenced by its ability to handle this message). This ensures that the node can re-insert itself into the network during the next sync action.
3. The node then removes each down node from its existingNodes set. For each metric type in the dataMap object, it removes the down nodes from the associated values. If a metric type has no remaining values after this deletion, the metric type is also removed from the dataMap.

By following these steps, the node ensures that its internal state reflects the current status of the network, removing any nodes that are down and thereby maintaining the efficiency and responsiveness of the system.


Remove itself from the downnodes list
Yes
Receive downnodes message
Check if itself is included in the list
No
For each metricType in dataMap remove from its values the down nodes
23. Diagram of “Down Nodes” flow

4.4 Code Overview

From a code perspective, the system's architecture comprises five principal components:

1. **Controllers**: These modules manage incoming requests at the endpoints of the system. Five distinct controllers—PodController, ContainerController, MetricController, LifecheckController, and DhtController—have been meticulously designed to address various facets of system functionality.
2. **Entities**: Serving as the structural backbone of the database, entities represent individual tables. The system embodies three entities—PodEntity, ContainerEntity, and MetricEntity—each delineating the data schema stored within the database.

3. **Repositories**: These repositories serve as intermediaries facilitating seamless communication with the database, enabling efficient retrieval and storage of data. The system is underpinned by three repositories—PodRepository, ContainerRepository, and MetricRepository—each meticulously crafted to manage interactions with its respective entity.
4. **Services**: The logic and functionality of the system are encapsulated within services, ensuring modular and maintainable code. Five meticulously engineered services—MetricService, LifecheckService, DhtService, PodService, and ContainerService—drive various system functionalities, adhering to best practices in software engineering.
5. **Kafka Integration**: Facilitating autonomous node operation, Kafka Integration is instrumental in enabling seamless communication and data distribution among nodes. Leveraging Apache Kafka, this component ensures fault tolerance and scalability, vital attributes in distributed systems architecture.

From a functional standpoint, the system is using four distinct tools:

1. **Database Integration**: This toolset encompasses database configuration and interaction, ensuring robust data storage and retrieval mechanisms. Seamlessly integrated within the system architecture, this component upholds data integrity and consistency, fundamental pillars in database management.
2. **Distributed Hash Table Implementation**: Enabling targeted requests and synchronization of information across nodes, the Distributed Hash Table Implementation is pivotal in facilitating seamless node initialization and internode communication. Its role in maintaining network coherence and facilitating efficient data exchange cannot be overstated.
3. **REST API**: Serving as the interface through which clients interact with the system, the REST API provides standardized communication protocols. Whether facilitating interactions between nodes or external clients, such as Postman, this component ensures seamless data exchange and interoperability.
4. **Kafka Integration**: This component harnesses the power of Apache Kafka to facilitate real-time data distribution and processing among nodes. By acting as consumers or subscribers to designated topics, Kafka Integration enhances system scalability and resilience, crucial attributes in contemporary distributed systems.

4.5 Code Description

Now let's move on to the description of the code. In this section, we will describe the most significant parts of the code, while the remaining methods and files will be detailed in the Appendix section.

4.5.1 Config Package

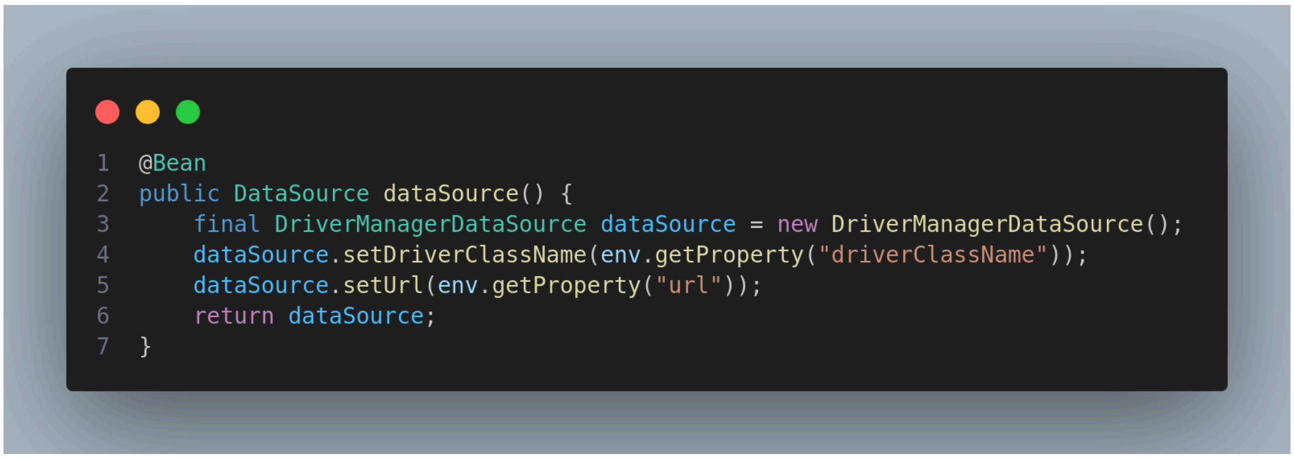
4.5.1.1 DbConfig.java

The DbConfig class is responsible for configuring the database connection and entity manager for the Spring Boot application. It defines the data source, entity manager factory, and Hibernate properties necessary for integrating with the database. This class plays a vital role in the configuration of the Spring Boot application, ensuring seamless integration with the underlying database. It is closely connected with the SpringSqliteApplication class, as the latter relies on the database configuration provided by DbConfig for initializing the application context and enabling data access. Also, there are two important class annotations:

- **Configuration**: This annotation indicates that the class contains bean definitions and should be processed by the Spring container during application context initialization.
- **EnableJpaRepositories**: This annotation enables JPA repositories for the application, specifying the base package where Spring should scan for repository interfaces. In our case, the target package is "com.thesis.sqlite.repositories"

The following methods were created and in order to initialize a Bean or as helper methods. A bean is simply a Java object that is instantiated, configured, and managed by the Spring IoC container. By default, beans are created as singletons within the container, meaning that there's only one instance of each bean shared throughout the application context. Beans can be injected into other beans or components within the application, facilitating loose coupling and modular design. Spring manages the lifecycle of beans, handling their creation, initialization, destruction, and disposal as needed.
Below, there are details about each method of DbConfig.java class:

- **dataSource**: This method creates and configures the data source for the application, setting the driver class name and database URL based on properties retrieved from the environment. Below is visible the implementation of this class



```java
@Bean
public DataSource dataSource() {
    final DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(env.getProperty("driverClassName"));
    dataSource.setUrl(env.getProperty("url"));
    return dataSource;
}
```

24. Implementation of dataSource Bean

- **entityManagerFactory**: This method configures the entity manager factory, specifying the data source, packages to scan for entity classes, and Hibernate vendor adapter. It also sets additional Hibernate properties retrieved from the environment.
- **additionalProperties**: This method constructs and returns a Properties object containing additional Hibernate properties retrieved from the environment, such as Hibernate dialect, SQL logging configuration, and schema generation strategy.

4.5.2 Entities Package

4.5.2.1 MetricEntity.java

This class is used in order to create a table named Metric in the database. This table will keep information about the metrics which are generated from the system.

It contains 10 fields:

- **id**: Id is the unique identifier of the container, and it is annotated with GeneratedValue annotation in order to auto create the value of this column based on the existing values
- **entityType**: It is the type of the metric and its available values come from the EntityTypeE class, which will be described later. It is annotated with column annotation in order to ensure that this column won't have null value
- **unit**: It is the unit of the metric value. Its available values are PERCENTAGE or PLAIN, and they came from UnitE enumeration which is an inner class in the MetricEntity class. The default value of this field is PLAIN.
- **minVal**: Is the minimum acceptable value for this metric. For example, when we run a CPU intensive process we can set this field to something big in order to be aware that if our actual value is lower than the minVal then something went wrong during this execution. This field is optional
- **maxVal**: It’s the opposite of the minVal
- **higherIsBetter**: This field shows us if the higher is the value of this metric, then the better for our system. For example, if the value of the free disk space is high, then this means that we have a lot of space left. The default value of this field is false
- **val**: This field has the actual value of the metric. It is annotated with column annotation in order to ensure that this column won't have null value
- **timestamp**: This field shows us the time when we received this metric. This can help up to determine the state of a node during a specific time period. It is annotated with column annotation in order to ensure that this column won't have null value
- **pod**: This field shows us if this metric came from a pod. It has a ManyToOne relationship with Pod table as we can have multiple metrics for each pod, but one metric can belong only to one pod
- **container**: This field shows us if this metric came from a container. It has a ManyToOne relationship with Containers table as we can have multiple metrics for each container, but one metric can belong only to one container

Finally, the class is annotated with lombok annotations for getter, setter, noArgsContructor and toString in order to minimize the code and increase the readability of it.

4.5.3 Controllers Package

4.5.3.1 MetricController.java

The MetricController class serves as a REST controller responsible for handling HTTP requests for adding, fetching or deleting metrics from the SQLite application. The controller is annotated with the following annotations:

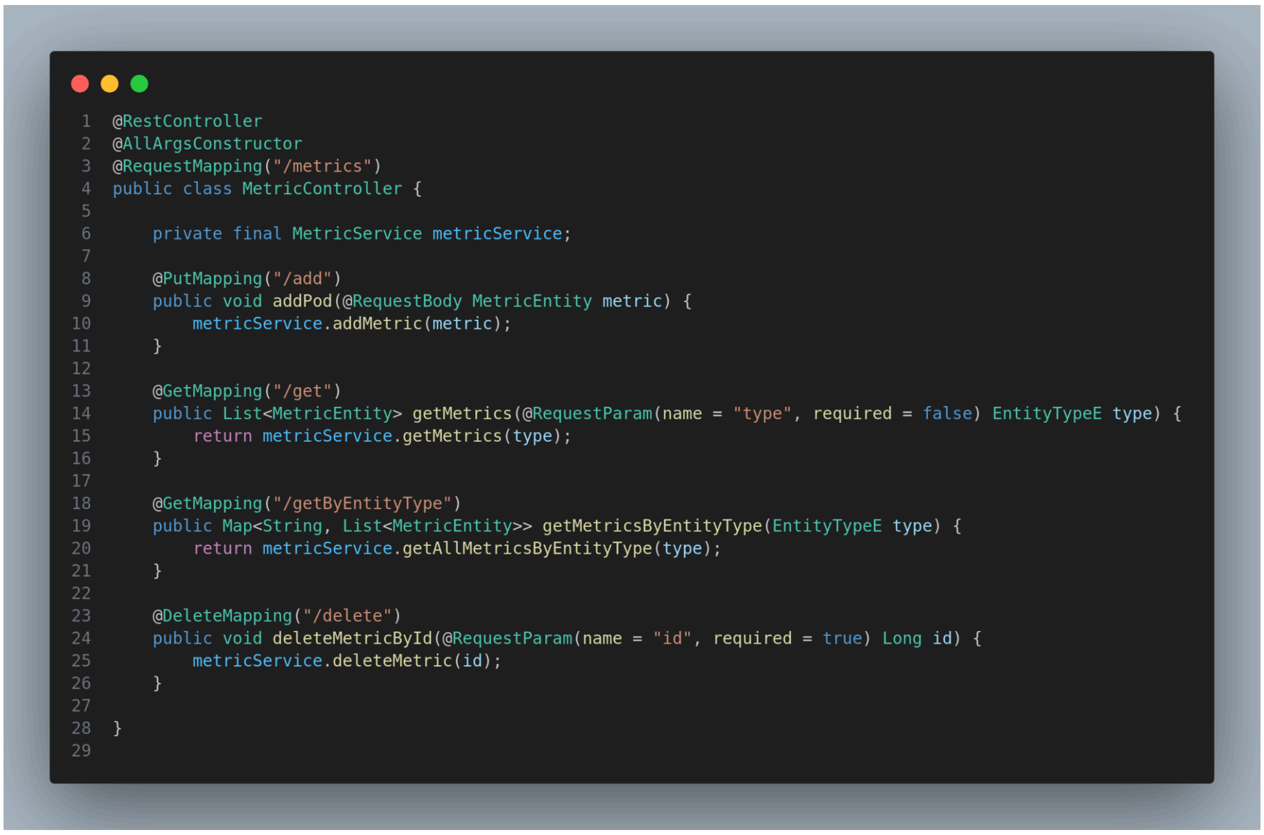
- **RestController**: Indicates that this class is a REST controller, allowing it to handle incoming HTTP requests and return responses.
- **AllArgsConstructor**: Lombok annotation that generates a constructor with arguments for the class, injecting dependencies into the controller.
- **RequestMapping("/metrics")**: Specifies the base URI path for mapping requests to this controller. That means that we are able to call the endpoint of this controller simply by sending a request to "http://localhost:{mapped-port}/metrics". Each defined endpoint can have additional parts for its URI, but it is always prefixed from the base URI.

This controller contains an autowired variable named metricService which is initialized from the constructor automatically. Also, it contains the following four methods:

- **addMetric**: Defines a new PUT endpoint which is used to add a new metric to the database. It expects a JSON payload representing a MetricEntity object in the request body. Upon receiving a request, it invokes the addMetric method of the MetricService to persist the metric entity. Finally, it adds one extra part in the URI so the final endpoint url is "http://localhost/{port-mapping}/metrics/add".
- **getMetrics**: Defines a new GET endpoint for retrieving all metrics from the database of the target node. It returns a list of MetricEntity objects representing all metrics stored in the database of the target node. This endpoint invokes the getMetrics method of the MetricService to fetch the metrics. Additionally, this endpoint optionally takes two request parameters:
  - "type" with type EntityTypeE: If this variable is present, then the endpoint returns only the metrics which have this entity type.
  - "node" with type String: If this variable is present, then the endpoint will return the metrics of the given node. In different case, it returns the metrics of the node who received the request.
  - Finally, it adds two extra parts in the URI so the final endpoint url is "http://localhost/{port-mapping}/metrics/get?type={type}&node={node}".

- **getMetricsByEntityType**: Defines a new GET endpoint for retrieving all metrics from the database of every node which is known in the target node. It returns a map which has as keys the hostnames of the nodes which returned results and a value of each key is a list of MetricEntity objects representing all metrics stored in the database of each node. This endpoint invokes the getAllMetricsByEntityType method of the MetricService to fetch the metrics. Additionally, it takes a mandatory argument of type EntityTypeE named "type". Based on this type, the application will determine where it should send request to retrieve data. In case a node does not return any metrics, then its key is removed from the final map. Finally, it adds one extra part in the URI, so the final endpoint url is "http://localhost/{port-mapping}/metrics/getByEntityType?type={type}".
- **deleteMetricById**: Defines a new DELETE endpoint used for deleting an existing metric from the database. It is mandatory to provide a request argument named "id" with the value representing the target metric's ID for deletion. Upon receiving a request, it invokes the deleteMetric method of the MetricService to delete the target metric entity. Additionally, it appends one extra part to the URI, resulting in the final endpoint URL being "http://localhost/{port-mapping}/metrics/delete?id={target-id}".

Below there is the code of MetricController in order to be used as reference for the terms which are used in controllers descriptions:



```java
@RestController
@AllArgsConstructor
@RequestMapping("/metrics")
public class MetricController {

    private final MetricService metricService;

    @PutMapping("/add")
    public void addPod(@RequestBody MetricEntity metric) {
        metricService.addMetric(metric);
    }

    @GetMapping("/get")
    public List<MetricEntity> getMetrics(@RequestParam(name = "type", required = false) EntityTypeE type) {
        return metricService.getMetrics(type);
    }

    @GetMapping("/getByEntityType")
    public Map<String, List<MetricEntity>> getMetricsByEntityType(EntityTypeE type) {
        return metricService.getAllMetricsByEntityType(type);
    }

    @DeleteMapping("/delete")
    public void deleteMetricById(@RequestParam(name = "id", required = true) Long id) {
        metricService.deleteMetric(id);
    }

}
```

25. Implementation of MetricController

4.5.3.2 LifecheckController.java

The LifecheckController class serves as a REST controller responsible for sending a “heartbeat” to its caller. By this way, we are able to check if a node is still up or something is wrong with this node. The controller is annotated with the following annotations:

- **RestController**: Indicates that this class is a REST controller, allowing it to handle incoming HTTP requests and return responses.
- **AllArgsConstructor**: Lombok annotation that generates a constructor with arguments for the class, injecting dependencies into the controller.
- **RequestMapping("/lifecheck")**: Specifies the base URI path for mapping requests to this controller. That means that we are able to call the endpoint of this controller simply by sending a request to “http://localhost:{mapped-port}/lifecheck”. Each defined endpoint can have additional parts for its URI, but it is always prefixed from the base URI.

This controller does not contain any variables. Also, it contains the following method:

- **lifecheck**: Defines a new GET endpoint which is used to check if the node is still alive. Each time someone is calling this endpoint, it means that the node failed to respond in time or the response produced an error. Given that, the endpoint is logging a warning message in order to inform the administrator of the system that something went wrong during the fetching of the data. Some potential reasons which can lead to this request is the amount of the data in the system, a failure in the network, the multiple requests which are currently handled by the system and the workload of the database or the whole node. After the log message, the endpoint is just returning true to the caller in order to inform him/her that the node is still up and running. Finally, this endpoint is reachable via “http://localhost/{port-mapping}/lifecheck” as the endpoint does not add any extra part.

4.5.4 Repositories Package

Repositories play a crucial role in simplification and maintainability of the code. They also allow us to be able to easily change the underlying database type. Every repository which will be described in this section is extending the CrudRepository interface [31]. This interface promotes code reusability, reduces boilerplate code, and enhances the maintainability of data access layers in Spring applications. The CrudRepository interface is part of the Spring Data framework and provides a set of standard methods for performing CRUD (Create, Read, Update, Delete) operations on entities in a database. It serves as a generic repository interface, allowing us to define custom repository interfaces for their entity classes without implementing CRUD methods manually. Some of its key capabilities can be found below:

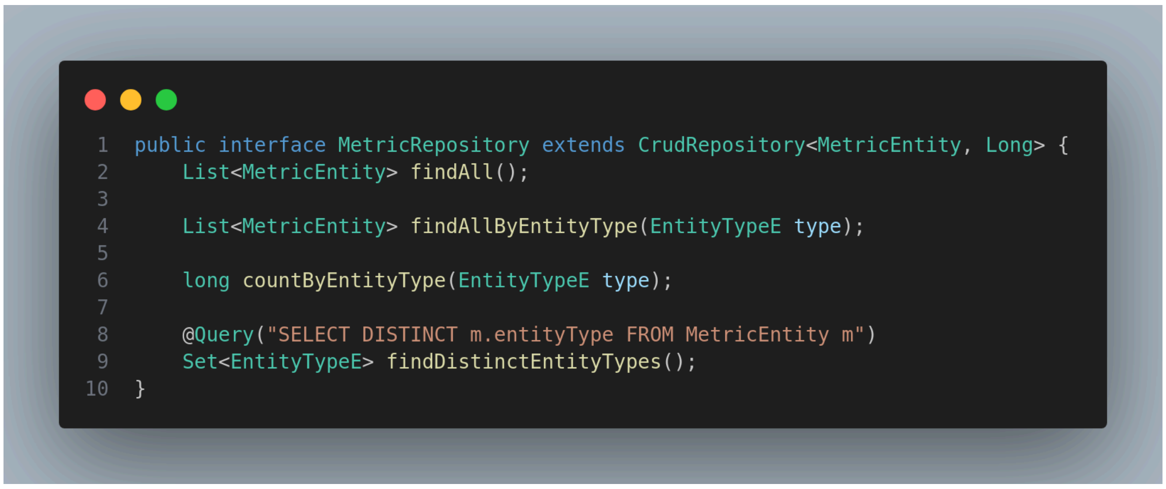
- **Create**: The interface includes methods for saving single entities (save(S entity)) and collections of entities (saveAll(Iterable<S> entities)), allowing new records to be added to the database.
- **Read**: We can retrieve entities by their primary key (findById(ID id)), retrieve all entities (findAll()), or retrieve entities by a set of primary keys (findAllById(Iterable<ID> ids)).
- **Update**: The CrudRepository interface provides methods for updating existing entities (save(S entity)), enabling modifications to entity attributes and relationships.
- **Delete**: We can delete entities by their primary key (deleteById(ID id)), delete a single entity (delete(T entity)), or delete multiple entities (deleteAll(Iterable<? extends T> entities)).

4.5.4.1 MetricRepository.java

The MetricRepository interface, designed as part of the Spring Data JPA part of our application and serves as an abstraction layer for database interactions related to MetricEntity objects within the SQLite application. It encapsulates methods for the retrieval and manipulation of metric data stored in the underlying database. This class contains the following 3 methods:

- **findAll()**: This method retrieves all MetricEntity instances stored in the database and returns them as a list. It offers a straightforward approach to fetch all metric entities without any filtering criteria. We create this method as we want a different return type than the type which is returned from findAll() method from CrudRepository.
- **findAllByEntityType(EntityTypeE type)**: This method retrieves MetricEntity instances from the database based on the specified entity type. It accepts an EntityTypeE enum parameter representing the desired type of metric entities to be fetched. This method is useful for fetching metric entities filtered by their entity type.
- **countByEntityType(EntityTypeE type)**: This method calculates and returns the count of MetricEntity instances in the database that match the specified EntityTypeE type. It provides a convenient way to obtain the total number of metric entities associated with a particular entity type. We use this method in order to determine whether we have metrics of a specific type saved in the database.
- **findDistinctEntityTypes()**: This method returns a set of the distinct entity types which exists in the database. It provides a convenient way to obtain the existing entity types in order to initialize the dataMap variable of DhtService for a node which has an existing database. With the usage of this method, we can avoid the fetching of every metric in order to determine the existing entity types.

Below you can find the MetricRepository implementation in order to clarify any potential gaps:



```java
public interface MetricRepository extends CrudRepository<MetricEntity, Long> {
    List<MetricEntity> findAll();

    List<MetricEntity> findAllByEntityType(EntityTypeE type);

    long countByEntityType(EntityTypeE type);

    @Query("SELECT DISTINCT m.entityType FROM MetricEntity m")
    Set<EntityTypeE> findDistinctEntityTypes();
}
```

26. Implementation of MetricRepository class

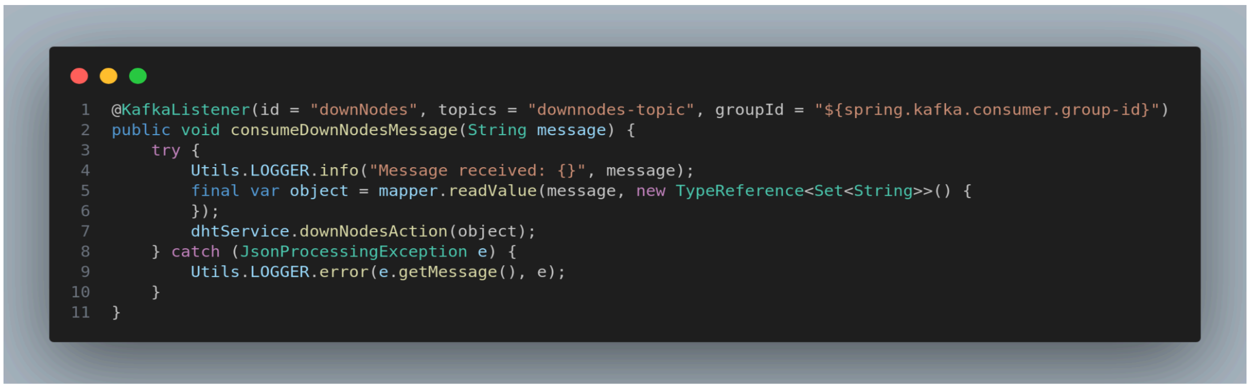
4.5.5 Kafka Package

4.5.5.1 KafkaConsumer.java

The KafkaConsumer class is a Spring service component responsible for consuming messages from Kafka topics within our application. It leverages the Spring Kafka integration to listen for messages on specific topics and process them accordingly. There are two methods in this class:

- **consumeInitNodeMessage**: This method is annotated with @KafkaListener and specifies the "init-topic" Kafka topic to consume messages from. It processes incoming messages by checking if the producer ID matches the ID of the producer service, ensuring the message is intended for consumption. If the conditions are met, the message is processed, deserialized into a BaseRequest object using the ObjectMapper, and passed to the DhtService for further handling. Note that every node has the same group ID, which is "group_id," to ensure that the message will be handled only by one node. This approach helps us avoid overloading the new node with sync requests from every other node, as the new node will receive only one sync request.
- **consumeDownNodesMessage**: This method is annotated with @KafkaListener and specifies the "downnodes-topic" Kafka topic to consume messages from. Upon receiving a message, it deserializes the message into a set of strings representing down nodes using the ObjectMapper and passes it to the DhtService for appropriate action. Note that in this case, every node has its own group ID, which is defined based on its hostname. This approach is followed to ensure that every node receives this message and proceeds with the necessary internal operations, which will be analyzed later.

An example of a Kafka consumer method is visible below:



```java
@KafkaListener(id = "downNodes", topics = "downnodes-topic", groupId = "${spring.kafka.consumer.group-id}")
public void consumeDownNodesMessage(String message) {
    try {
        Utils.LOGGER.info("Message received: {}", message);
        final var object = mapper.readValue(message, new TypeReference<Set<String>>() {
        });
        dhtService.downNodesAction(object);
    } catch (JsonProcessingException e) {
        Utils.LOGGER.error(e.getMessage(), e);
    }
}
```

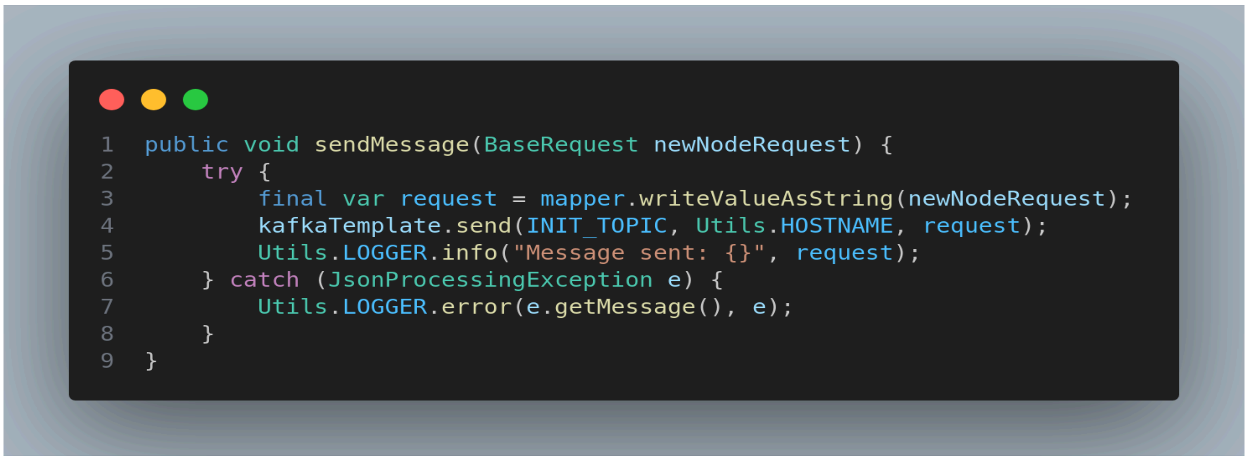
27. Implementation of consumer method for the consumption of “downNodes” messages

4.5.5.2 KafkaProducer.java

The KafkaProducer class is a Spring service component responsible for producing messages to Kafka topics within our application. It utilizes the Spring Kafka integration to interact with Kafka brokers and send messages to specific topics. This class container two methods:

- **sendMessage**: This method sends a message containing a BaseRequest object to the "init-topic" Kafka topic. It serializes the BaseRequest object into a JSON string using the ObjectMapper and sends it to the Kafka topic via the KafkaTemplate. It logs the sent message for tracking purposes.
- **sendDownNodesMessage**: This method sends a message containing a set of strings representing down nodes to the "downnodes-topic" Kafka topic. Similar to the previous method, it serializes the set of strings into a JSON string using the ObjectMapper and sends it to the Kafka topic via the KafkaTemplate. It also logs the sent message.

An example of a Kafka producer method is visible below:



```java
public void sendMessage(BaseRequest newNodeRequest) {
    try {
        final var request = mapper.writeValueAsString(newNodeRequest);
        kafkaTemplate.send(INIT_TOPIC, Utils.HOSTNAME, request);
        Utils.LOGGER.info("Message sent: {}", request);
    } catch (JsonProcessingException e) {
        Utils.LOGGER.error(e.getMessage(), e);
    }
}
```

28. Implementation of Kafka producer for “init-node” messages

4.5.6 DHT Package

The distributed hash table (DHT) is a very important component of our application. This component is responsible for the synchronization between the nodes of the network. Our current implementation is a distributed hash table which is replicated in every node. An idea for future work in our application is to try different approaches about the DHT architecture.

4.5.6.1 DhtController.java

The DhtController class serves as a REST controller responsible for receiving requests for new nodes, syncing or delete nodes. It is the entry point of the synchronization mechanism of our application. The controller is annotated with the following annotations:

- **RestController**: Indicates that this class is a REST controller, allowing it to handle incoming HTTP requests and return responses.
- **AllArgsConstructor**: Lombok annotation that generates a constructor with arguments for the class, injecting dependencies into the controller.
- **RequestMapping("/dht")**: Specifies the base URI path for mapping requests to this controller. That means that we are able to call the endpoint of this controller simply by sending a request to “http://localhost:{mapped-port}/dht”. Each defined endpoint can have additional parts for its URI, but it is always prefixed from the base URI.

This controller autowires the dhtService variable which is initialized in the constructor of the class. Also, it contains the following three methods/endpoints:

- **initializeNode**: Defines a new PUT endpoint which is used to add a new node to the system. It expects a JSON payload representing a BaseRequest object in the request body. Upon receiving a request, it invokes the addNode method of the DhtService to add the new node to the DHT of the node and to proceed with the necessary actions. Finally, it adds one extra part in the URI so the final endpoint url is “http://localhost/{port-mapping}/dht/init”.
- **syncNode**: Defines a new PUT endpoint which is used to sync the data of this node with the sender of the request. It expects a JSON payload representing a SyncRequest object in the request body. Upon receiving a request, it invokes the receiveData method of the DhtService to sync its DHT with the incoming data. Finally, it adds one extra part in the URI so the final endpoint url is “http://localhost/{port-mapping}/dht/sync”.
- **downNodes**: Defines a new PUT endpoint which is used to inform the node that some nodes are down. It expects a set of string payload representing the hostnames of the nodes which are down. Upon receiving a request, it invokes the downNodesAction method of the DhtService to remove the down nodes from the DHT. Finally, it adds one extra part in the URI so the final endpoint url is “http://localhost/{port-mapping}/dht/down”.

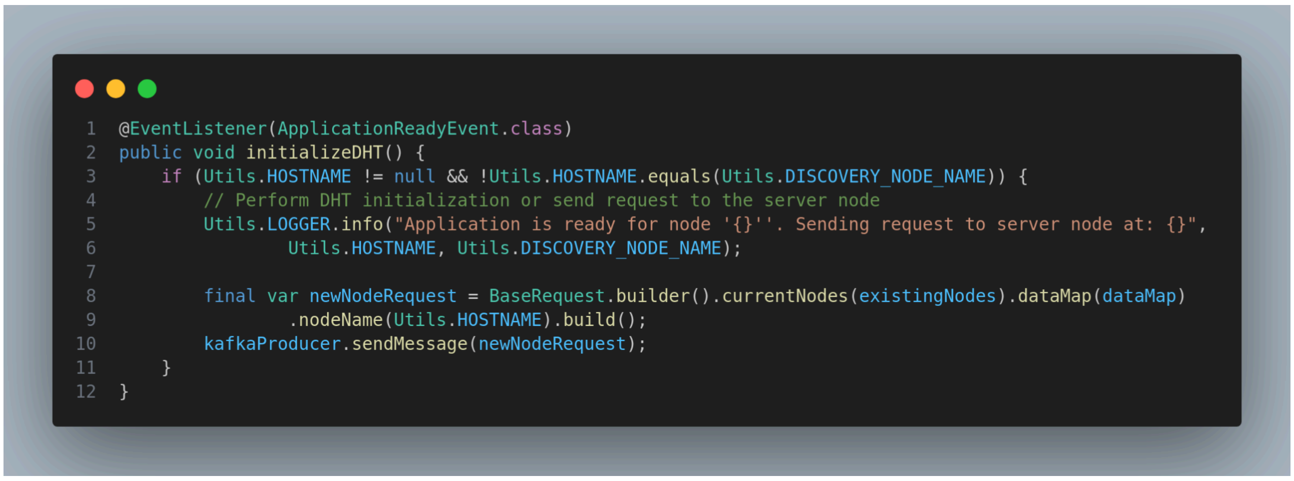
4.5.6.2 DhtService.java

The DhtService class is a central component responsible for managing the distributed hash table (DHT) within the application. It handles the storage, synchronization, and communication of data among different nodes in the system. The class utilizes synchronized blocks and locks (deleteLock and syncLock) to ensure thread safety during data modification and synchronization operations. Upon instantiation, the DhtService initializes various data structures and dependencies required for its operation, such as the data map, deleted types map, Kafka producer, and REST template. As the variables of this service are used for a specific purpose, let's analyze them one by one:

- **dataMap**: This map is used in order to be aware of the nodes which contain a specific entity type. It is vital for our system in order to limit the request to the lowest possible. In that way, we can lower the pressure in our network.
- **deletedTypesMap**: This map is used in order to be aware of the nodes which no longer have a specific entity type. Due to syncing actions, we might receive an invalid info e.g. that a node has a specific entity type while we had just received another message that this entity type has been removed from the target node. This is happening as everything is asynchronous, so the node who is sending the sync action has not received the message for the deletion yet. In order to ensure that we will have the correct nodes, when we receive a deletion message, we add this pair to deletedTypesMap. As long as this pair exists in this map, we will not add this pair to the dataMap if it is from a syncing action. We will analyze this further in the description of the methods.
- **deletedTypes**: This set contains the entity types which have been deleted from a node between the previous syncing actions and the present. We send this set to the other nodes during syncing action and then we clean up the map.
- **restTemplate**: This variable is used in order to be able to send request to our rest API
- **existingNodes**: This is a set which keeps every node which is known from the specific instance of our application. In our current implementation, the usage of Kafka makes the usage of this field less frequent, as we use it only during syncing actions. In a previous version (which can be used as fallback) of our system is using rest API requests instead of Kafka messages, so this variable is used more frequently in order to send the request to every node.
- **kafkaProducer**: This variable is used in order to send messages through Kafka
- **deleteLock**: This variable is used in order to achieve synchronization during actions related with the deletion of a node. Synchronization helps us to achieve consistency in our application and to avoid potential errors and race conditions.
- **syncLock**: This variable is used in order to achieve synchronization during actions related with the synchronization of our nodes. Synchronization helps us to achieve consistency in our application and to avoid potential errors and race conditions.

Additionally, we might face cases where a node might was previously active, but it went offline at some point. In those cases, when the node goes online again, during the initialization of dhtService we initialize the data map by adding the entity types which are present in the database of the node. Apart from the constructor, this class has 13 methods. The details of each method can be found below:

- **initializeDHT()**: This method is triggered by the ApplicationReadyEvent. It constructs a BaseRequest object and sends it through Kafka to the other nodes. When a node catches this message, it performs a syncing with the new node in order to update it with the existing data in our system. More about this process will be analyzed in the addNode method. Kafka allows the system to be agnostic when it comes to the existing nodes in the system. Additionally, we add the server node to the known nodes of the new node to ensure that even if the kafka message is lost, then the node will eventually share its data with the rest network. The code can be seen below:



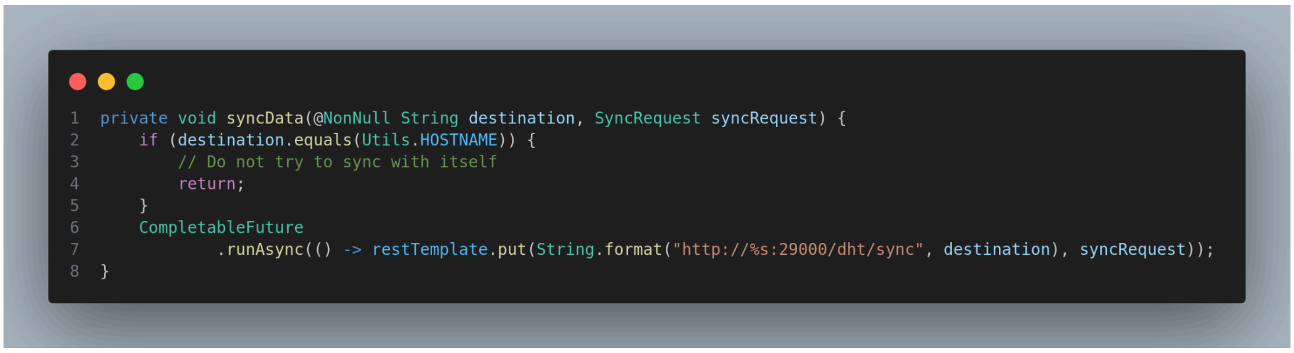
```java
@EventListener(ApplicationReadyEvent.class)
public void initializeDHT() {
    if (Utils.HOSTNAME != null && !Utils.HOSTNAME.equals(Utils.DISCOVERY_NODE_NAME)) {
        // Perform DHT initialization or send request to the server node
        Utils.LOGGER.info("Application is ready for node '{}''. Sending request to server node at: {}",
                Utils.HOSTNAME, Utils.DISCOVERY_NODE_NAME);

        final var newNodeRequest = BaseRequest.builder().currentNodes(existingNodes).dataMap(dataMap)
                .nodeName(Utils.HOSTNAME).build();
        kafkaProducer.sendMessage(newNodeRequest);
    }
}
```

29. Implementation of "initializeDHT" method

- **put(EntityTypeE key, Set<String> value, boolean fromSyncAction)**: This method adds data to the node's data map with the specified key and value. First, it ensures thread safety using synchronization with the deleteLock. Then it checks if the request came from sync action or due to a new entry to the database. When the request comes from a sync action, then we use the deletedTypesMap variable to remove every node which seems to be deleted. In different case, if the request is due to a new entry to the database then we check if the entity type of the new entry is present in the deletedTypes variable and if so we remove it from the map to avoid sending invalid information. We also remove it from deletedTypesMap to avoid skipping of this node during fetching of the metrics. In this stage, we have ensured the consistency for the potentially deleted nodes so we proceed by adding the new entry to the dataMap variable. Node that if the specified key is already present in the map and its value (a list of strings) contains the HOSTNAME then the map is remaining the same.

- **get(EntityTypeE key)**: This method retrieves data from the node's data map based on the provided key. It returns a set of strings representing the hostnames associated with the specified entity type. If the specified entity type is missing from the dataMap then we just return an empty list. Note that there is no point for synchronization in this point as dataMap is an instance of ConcurrentHashMap.
- **putDataToNode(EntityTypeE key)**: This method is used in order to add a new metric to the database of the node and it is called from the MetricService class. We use a synchronization block in order to ensure consistency and we call the put() method to handle the rest actions.
- **deleteTypeFromNode(EntityTypeE key)**: This method is used during the deletion of a metric from the database, if there are no more metrics of this type in the database. The body of the method is executed inside a deleteLock synchronization block. First, it removes the specified type of data from the node's data map and marks it for deletion. Then, it updates the deleted types map to track deleted data.
- **addNode(BaseRequest newNode)**: This method is used when a Kafka message for a new node is handled for this node. By executing the receiveData method of the dhtService, It adds the new node to the DHT and inserts its data to the required variables (dataMap, existingNodes and deletedTypesMap). Then a sync request is sent to the new node which contains the information of the current node. By doing this, we ensure that the new node, after the sync action, contains the updated data so it can return a response which will contain the expected metrics.
- **syncData(String destination, SyncRequest syncRequest)**: This method is used in order to send the sync request to a specific node. First we check that the destination node is not the current node and if so we send the request. Node that we utilize the usage of threads in order to decouple the execution of the request with the execution of our main program. The code can be seen below:

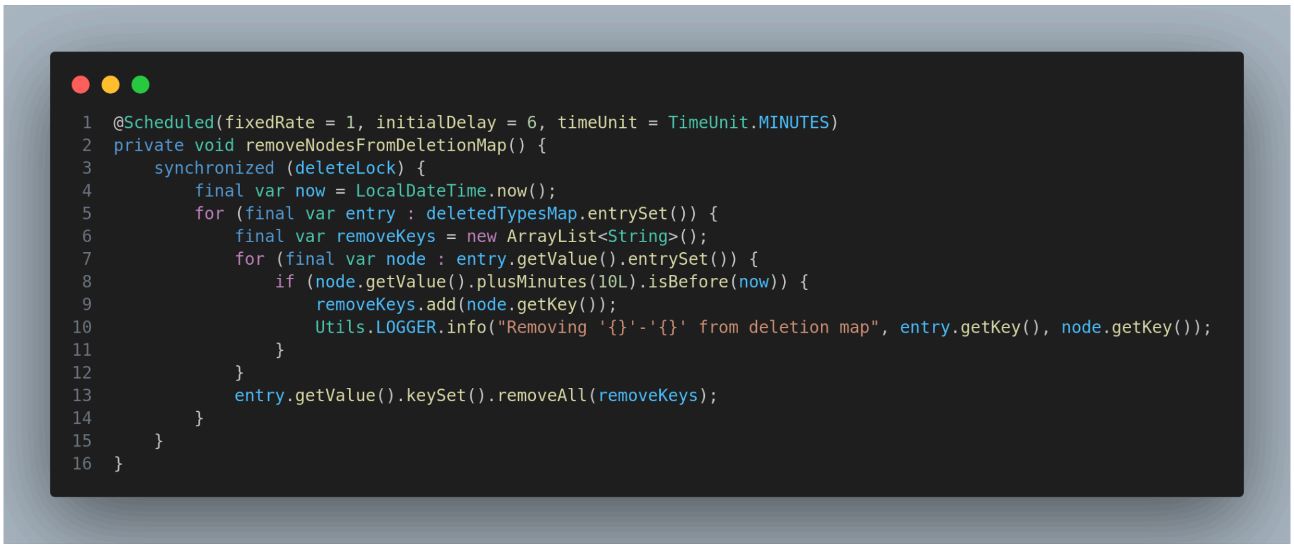


```java
private void syncData(@NonNull String destination, SyncRequest syncRequest) {
    if (destination.equals(Utils.HOSTNAME)) {
        // Do not try to sync with itself
        return;
    }
    CompletableFuture
            .runAsync(() -> restTemplate.put(String.format("http://%s:29000/dht/sync", destination), syncRequest));
}
```

30. Implementation of “syncData” method for sending of sync data to a node

- **syncScheduledJob()**: This method is scheduled to run periodically for the DHT maintenance. In our configuration, the sync action is running every 5 minutes, but this should be changed depending on the usage of the application. The method is creating the syncRequest object in a synchronization block to ensure consistency, and then it iterates through the known nodes to send the sync requests.

- **removeNodesFromDeletionMap()**: This method is scheduled to run periodically (every 1 minute) in order to remove nodes from the deletion map. This is happening in order to ensure that if a deleted entity type is re-inserted in a node then eventually every node will be aware about this change. The value of deletedTypesMap variable is a Map<String, LocalDateTime> where each per shows which node is deleted on what time. In this method, we fetch the current dateTime of our system and compare it with the value of the inner map. If more than 10 minutes has passed then we remove this entry from the map to allow the re-insertion of the entry in the dataMap. The time period which has passed before the deletion is related with the interval of the sync actions. In different case, we wait 2 cycles of syncing before removing it from the map. The code can be seen below:



```java
@Scheduled(fixedRate = 1, initialDelay = 6, timeUnit = TimeUnit.MINUTES)
private void removeNodesFromDeletionMap() {
    synchronized (deleteLock) {
        final var now = LocalDateTime.now();
        for (final var entry : deletedTypesMap.entrySet()) {
            final var removeKeys = new ArrayList<String>();
            for (final var node : entry.getValue().entrySet()) {
                if (node.getValue().plusMinutes(10L).isBefore(now)) {
                    removeKeys.add(node.getKey());
                    Utils.LOGGER.info("Removing '{}'-'{}' from deletion map", entry.getKey(), node.getKey());
                }
            }
            entry.getValue().keySet().removeAll(removeKeys);
        }
    }
}
```

31. Implementation of scheduled method “removeNodesFromDeletionMap”

- **buildSyncRequest()**: This method is used in order to build a sync request. In this process, we copy the dataMap, deletedTypes and existingNode variable into new objects to decouple them from the base object. This allows us to execute syncing without synchronization blocks.
- **receiveData(BaseRequest)**: This method is used in order to update the inner variable with the data which came to the node through a syncing or an add new node action. Every action is executed in a synchronization block with syncLock variable. First we update the existing nodes and we use the put method in order to update the dataMap variable. Then, if the request came from a syncing action, we open a synchronization block on deleteLock variable and we proceed with the update of deletedTypesMap. Additionally, if the deleted types exist in the dataMap variable then we remove them from the map.
- **downNodesAction(Set<String> downNodes)**: When a Kafka message about down nodes is received, then we execute this method. First we check if the current node was added by mistake in this list and if so we remove this node from the list. Then we remove the rest nodes from the existingNodes list and dataMap variable. We do this action in order to prevent delays in data access.

- **sendDownNodesRequest(Set<String> downNodes)**: When we determine that some nodes are down, we send a Kafka message which will be caught from every other node in the network in order to remove those nodes from their variables.

4.5.7 Services Package

4.5.7.1 MetricService.java

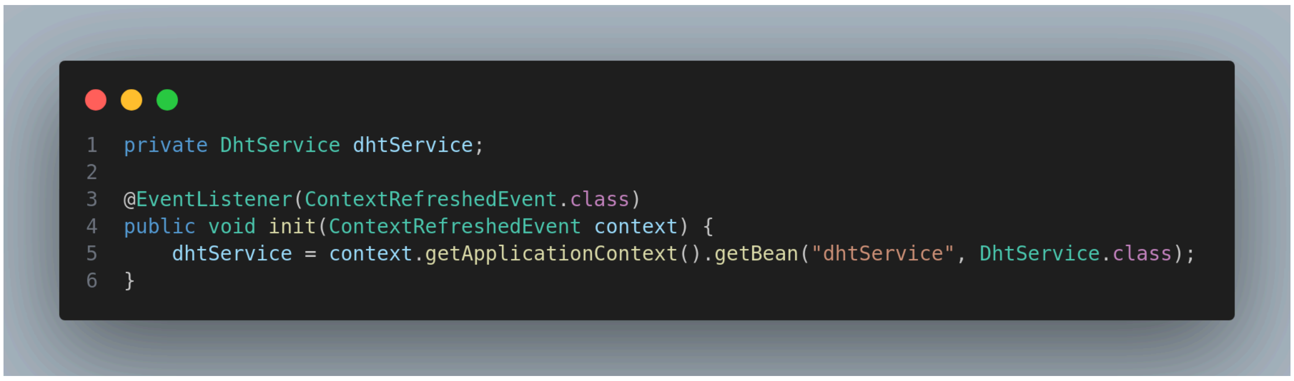
The MetricService class plays a crucial role in managing metrics within the application, providing functionalities for adding, deleting, and retrieving metrics, as well as orchestrating data distribution across the distributed hash table (DHT). The class is annotated with @Service, indicating that it is a service component managed by Spring. Additionally, the @AllArgsConstructor annotation generates a constructor with a parameter for each field in the class (metricRepository, podService, containerService, lifecheckService), enabling dependency injection of the required services. To avoid cycles in the dependencies of the services, we have created a new method named init. This method is annotated with @EventListener and listens for the ContextRefreshedEvent, which is triggered when the application context is initialized or refreshed. Upon initialization, it retrieves the DhtService bean from the application context and initializes the dhtService field. Also, this class contains the following 4 methods:

- **addMetric**: This method adds a new metric to the system. It first ensures that associated pods and containers are created or retrieved from the database using the PodService and ContainerService. The metric is then saved to the database, and its data type is distributed across the DHT framework using the putDataToNode method of the DhtService.
- **deleteMetric**: This method deletes a metric from the system based on its ID. It first retrieves the metric from the database and checks if it exists. If found, the metric is deleted from the database, and if no other metrics of the same type exist, the data type is removed from the DHT framework using the deleteTypeFromNode method of the DhtService.
- **getMetrics**: This method retrieves metrics from the database. If a specific type is provided in the parameters, then only metrics of that type are retrieved. Otherwise, all metrics are returned. It also takes an argument for the target node. If a target node is provided and based on the meta information the node exists and it contains metrics of the provided type, then we forward the request to the target node and return the metrics which exist in its database. In different case, we return the requested metrics from the node which received the initial request.
- **getAllMetricsByEntityType**: This method retrieves metrics of a specific type grouped by their source node. It first retrieves the nodes which have metrics of the requested type. Then, it asynchronously sends requests to each node to fetch the corresponding metrics data. Metrics data retrieved from each node is aggregated into a map, where the keys represent node names, and the values represent lists of metrics entities. Additionally, the method

handles timeouts and exceptions gracefully, ensuring robustness and reliability in data retrieval operations. If a node delays more than 4 seconds to respond, then we ignore its return value, and we add this node to a list in order to proceed with extra actions for its health. Currently, those actions have not been implemented. Before returning the result to the caller, if we have nodes which threw an error. We call asynchronously the lifecycle method from lifecycleService in order to check if those nodes are actually down or something else happened, e.g., a temporary network issue.

4.5.7.2 LifecheckService.java

The LifecheckService class is responsible for performing periodic health checks on the nodes within the system and taking appropriate actions based on the results. It ensures the availability and reliability of the nodes by detecting and handling any potential failures or downtimes. The class is annotated with @Service, indicating that it is a service component managed by Spring. Additionally, the @NoArgsConstructor annotation generates a constructor without parameters. Also, in order to avoid cycles in the service dependencies we have created a method named init. This method is annotated with @EventListener and listens for the ContextRefreshedEvent, which is triggered when the application context is initialized or refreshed. Upon initialization, it retrieves the DhtService bean from the application context and initializes the dhtService field. This method is visible below:



```java
private DhtService dhtService;

@EventListener(ContextRefreshedEvent.class)
public void init(ContextRefreshedEvent context) {
    dhtService = context.getApplicationContext().getBean("dhtService", DhtService.class);
}
```

32. Usage of ContextRefreshedEvent to avoid cycle in dependencies during startup

This class contains the following method:

- **lifeCheck**: It takes a set of node addresses as input and iterates over each node, sending a request to the /lifecheck endpoint to determine if the node is actually down. If a node fails to respond within a specified timeout period, it is considered down, and its address is added to the set of down nodes. Additionally, every node which is detected as down, is removed from the distributed hash tables (DHTs) to ensure data consistency and reliability. This is achieved by calling the sendDownNodesRequest method of the DhtService, which initiates the removal process for the down nodes. This method is called when during the fetching of some metric, there are nodes which throw an error.

5 Experimental analysis

The tests described in this thesis were conducted on a machine running Windows 11 operating system. The hardware configuration of the machine includes an AMD Ryzen 7 2700 Eight-Core Processor, 32GB of RAM operating at 2800 MHz, and an SSD storage device with a capacity of 224GB (WD Blue SSD, model WDS240G2G0A-00JH30). Additionally, the machine is equipped with an NVIDIA GeForce RTX 2060 GPU with 6GB of VRAM. This machine serves both as a testing environment and for personal use. It is a physical server, and Docker was utilized for virtualization purposes. Throughout the testing period, the machine experienced no maintenance activities or interruptions.

5.1 Usage Demonstration

In order to create the necessary resources and to start our application, the following steps must be followed:

1. **Remove Existing Docker Image**: Before starting the framework setup, ensure any existing Docker images related to the thesis are removed to maintain a clean environment. To do so, execute: *docker rmi -f edge-sqlite-db:1.0.0*
2. **Clean and Package the Maven Project**: Use Maven to clean the project, removing any previous build artifacts, and package it to generate the executable JAR file needed for deployment. To do so, execute: *mvn clean package* in the home directory of the framework.
3. **Build Docker Image**: Build a new Docker image incorporating the latest changes and configurations using the provided Dockerfile. To do so, execute: *docker build -f Dockerfile . -t edge-sqlite-db:1.0.0*
4. **Start Kafka and Zookeeper Services**: Initiate Kafka and Zookeeper services using the docker-kafka-compose.yml file to set up the necessary messaging infrastructure. To do so, execute: *docker compose -f docker-kafka-compose.yml up*
5. **Start SQLite Server and Related Services**: Launch the SQLite server and related services using the docker-compose.yml file to establish the required database environment. To do so, execute: *docker compose -f docker-compose.yml up*
6. **Start Client Containers**: Start client containers using the docker-client-compose.yaml file to simulate multiple client instances interacting with the system. To do so, execute: *docker compose -f docker-client-compose.yaml up*
7. **Start Second Client Container**: Add additional client containers using the docker-second-client-compose.yaml file to expand testing scenarios and simulate varying workloads. To do so, execute: *docker compose -f docker-second-client-compose.yaml up*

5.2 Postman Collection for Testing

A Postman collection is provided, encompassing a variety of requests and actions meticulously crafted to validate the system's functionality and performance. This collection serves as a valuable resource for conducting thorough testing and evaluation of the framework.

5.3 Testing of the implementation

In order to test the performance of our application as well as the validity of the responses sent from the system, we proceed with the following test cases:

- Started Kafka services, initiated one node set as the server, added metrics through Postman requests, and then started one more node. The new node sent a Kafka message, which was handled by the first node, which then sent back an abstract description of its data. We then added metrics to the new node and performed requests for metrics of type MEMORY_USAGE. When we sent the request to the first node, only its metrics were returned, as synchronization had not been performed yet. When we sent the request to the second node, metrics from both nodes were returned, as the inner map was populated with the correct information due to the initial syncing.
- Waited a few minutes until syncing took place. After syncing, we resent the requests to both nodes. The results returned were identical.
- Started one more node and checked the Kafka message. This time, the second node caught the message and performed the syncing with the new node. Requests to the new node yielded the same results as the other two. Then, we shut down the second node and resent the request from the new node. The result now was that the second node failed to respond. Then, a life check request was sent to the second node, and when it failed to respond, a Kafka message was sent to every node (first and third). Then, the second node was identified as down. We resent the request and verified that the second node was skipped from the requests.
- Then, we restarted the second node. The node fetched its data from the previous database and sent a Kafka request. The third node caught it and sent the sync request. Until syncing, only the second node and the third were returning the full results as expected. After the sync, the first node updated its map with the data from the second node, and the results of the requests were correct.
- We added a new metric to the third node with type CPU_USAGE. Requests to the first and second nodes weren't sent at all, as they didn't have info about it. Requests to the third node yielded the expected results. After syncing, the other two nodes returned the correct result. Also, using the debugger, I checked that only one request was sent to the third node and the other two nodes were skipped as expected.
- Performed deletion of the new metric in order to remove every metric with type CPU_USAGE from the third node. Waited until syncing and checked that the information about CPU_USAGE was deleted from the maps of the other nodes. Then, re-added the

metric to the node and sent a request about it. The third node responded as expected. During sync, the information about the CPU_USAGE was sent to the first and second nodes, but it was ignored as the deletion constraint was active. The same behavior happened again in the second sync request, but in the third one (after 10 minutes), the information was added to the map, and the other two nodes were able to respond to the request for the CPU_USAGE as expected.

- Started two more nodes, added a few metrics with various types, and performed different combinations of the tests described before. Everything worked as expected as nodes were sync as expected and the correct results were returned every time.
- Finally, stopped every container, apart from Kafka related containers, and started a client node. Send a request for data and then start the server node. The server node sent the Kafka request and received a sync request from the client node in order to archive the fast warm up of the node. Then send a request for the data to the server node and it returns data from both nodes as expected.

Through the initial testing of our application, we determined the following additional test cases in order to stretch our system and test its performance:

- Try starting multiple containers with the same compose file. This allows us to check if we have any issues when the same action is executed simultaneously from different nodes.
- Send many requests to check system load. The transmission of data over the network could lead to potential performance issues in our system. Also, in this case, the CPUs of the system are crucial as we need one virtual thread per request.
- Try to have consecutive addition of data while someone is trying to fetch data constantly. This will show us if there are performance issues due to synchronization points.
- After a short implementation in order to allow new types of requests, we can check how the system performs frequent requests in order to fetch the new data which have been inserted to the system.

5.4 Performance and Time Comparison Tests

To thoroughly evaluate the performance of our system, we conducted a series of tests focusing on time comparisons across various scenarios. These tests were designed to measure the efficiency of data querying, synchronization, and data fetching under different conditions. The test cases are as follows:

- **Query Time Comparison Between Nodes**: To evaluate the performance of the system when requesting data for a specific node compared to the performance of requesting the same data from the source node, we proceeded with the following experiment:
    1. Send a query request from Node A to Node B and record the response time.
    2. Send a query request directly to Node B and record the response time.
    3. Compare these times to evaluate the overhead of cross-node querying.
    4. Increased the size of the database in Node B and repeated the experiment.

This experiment was conducted for different sizes in the database of Node B to determine how the size affects the performance of the system. The results of the experiment are visible in the following table:

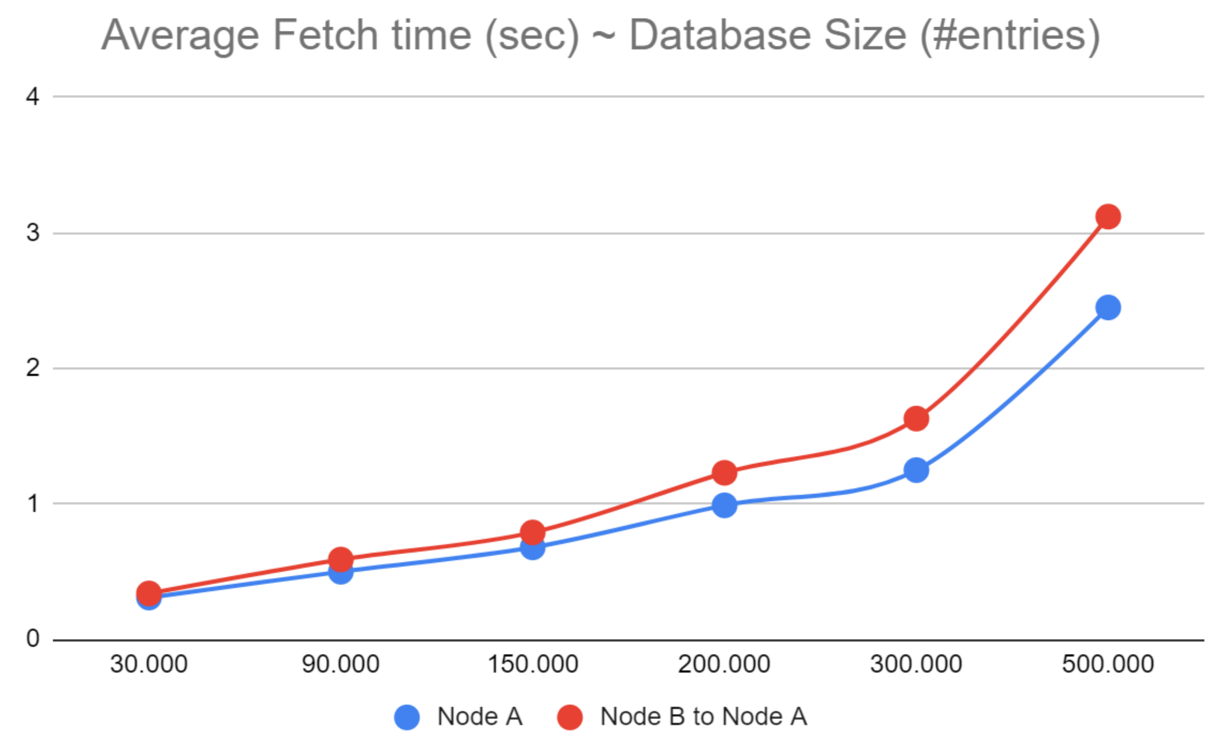
Entries count in Node A	Query Time in seconds (Node A)	Query Time in seconds (Node B to Node A)	Overhead (seconds)
30.000	0.31	0.34	0.03
90.000	0.50	0.59	0.09
150.000	0.68	0.79	0.11
200.000	0.99	1.23	0.24
300.000	1.25	1.63	0.38
500.000	2.45	3.12	0.67

1. Comparison of average query time with different number of entries and different target nodes from the request

The results of the query time comparison experiment reveal several key insights into the performance of the system when querying data from different nodes, particularly focusing on the overhead introduced by cross-node querying and the impact of database size on response times.

1. The experiment demonstrated that request data for Node A through Node B incurs a noticeable overhead compared to fetching directly from Node A. This overhead increases as the size of the database grows. Note that 1 Mb of data is approximately 5.000 entries. So, it is expected to notice this behavior as the data has to be transmitted over the network two times. Note that in our experiment, Node A and Node B were in the same machine so there is no network communication cost.
2. The response times for both direct and cross-node queries increase with the size of the database. This indicates that larger database sizes impose a heavier load on the system, resulting in longer query times. This is expected as the database will take longer to return the data and more information has to be transmitted over the network.
3. An Out Of Memory (OOM) exception occurred in Node B when dealing with the largest dataset size. This happened due to the fact that the framework is temporarily saving the data from the network into a Java object and then it is returned to the

caller. This is happening in order to be able to handle the intermediate data if necessary. Although, this indicates that the system may require better memory management techniques or more resources to handle extensive data volumes effectively. Note that the returned object in the database with the 500.000 entries was almost 100 Mb which was way too much for the 256Mb Xmx of the Java service.



33. Line chart with average fetch time (sec)

- **Sync Time Based on the Number of Nodes**: To measure the synchronization time as the number of nodes in the network increases, we performed the following steps:
    1. Incrementally add nodes to the network.
    2. Measure the time taken for a complete synchronization cycle with each additional node.

The results of this experiment are presented in the table below:

Number of Nodes	Sync Time (ms)
2	0.72

3	0.68
5	0.33
10	0.45

2. Sync time for different number of nodes

The results of the synchronization time experiment reveal that the sync time is not relative with the number of the nodes. This is happening due to the fact that the synchronized calls take place asynchronously. Additionally, it is clear that the time which is required for the sync is too small and this is the reason why we don't have consistent times.

- **Query Time for Specific Data at Different Sync Intervals**: To evaluate the impact of synchronization intervals on query performance (equivalent with the increase of the number of the nodes in the network), we carried out the following procedure:
    1. Performed 1000 queries for the data
    2. Decreased the interval time and retried
    3. Record and compare the response times to determine if synchronization intervals affect query efficiency.

The results are summarized in the table below:

Sync Interval (sec)	Average Query Time (sec)
30	1.01
15	1.02
10	1.02
5	1.03
2	1.03
1	1.04
0.5	1.04

3. Average query time for different sync intervals

The node used in this experiment had 200,000 entries, which provides a substantial amount of data for testing. Despite the large database size, the synchronization interval changes did not significantly affect the query performance. The experiment shows that varying the synchronization intervals has a minimal impact on the average query time. The differences in query times across different sync intervals are negligible. Finally, as a result of this experiment, we can conclude that the synching time does not have a significant effect on the performance of the system, at least for a few thousands of nodes.

- **Fetch Time for Different Numbers of Nodes**: To measure how the fetch performance is influenced by the number of nodes in the network, we performed the following steps:
    1. Execute fetch operations for specific metric type from every node
    2. Record the response time
    3. Increased the number of nodes and retried the experiment
    4. Analyze how the number of nodes impacts fetch efficiency.

  The findings of this experiment are shown in the table below:

Number of Nodes	Average Fetch Time (ms)
2	307
3	300
5	320
10	324

4. Average fetch time for different number of nodes

The results indicate that while there is a slight increase in fetch time as the number of nodes increases, the impact is relatively small. For example, the increase from 2 to 10 nodes results in an additional fetch time of only 17 milliseconds. The reason for this could be minor delays in the network of a node or in the database of a node. It is expected that as more nodes are in the system the probability of a delay in a node increases. These findings suggest that the fetch performance is reasonably robust and scalable, handling additional nodes with only minor increases in response times. This indicates that the system is designed to efficiently manage data fetching across multiple nodes, maintaining performance even as the network grows. This characteristic is important for ensuring that the system can scale effectively while maintaining acceptable performance levels for fetch operations.

6 Conclusions & Future Work

Throughout this thesis, we conducted a thorough analysis of edge computing, highlighting its crucial role in addressing the growing challenges posed by the increasing data from IoT devices. By strategically placing computation and storage closer to where data is produced, edge computing offers clear benefits such as faster response times, better use of bandwidth, and improved data privacy. We carefully examined both the advantages and disadvantages of edge computing, delving into the complexities of security issues and vulnerabilities that may arise within an edge system. Additionally, we conducted a detailed survey of various databases suitable for different edge computing systems, assessing their suitability for our application. Following this, we developed an innovative framework using SQLite for distributed telemetry queries. Our explanation covered the fundamental tools supporting the proposed framework, including Docker, Kafka, and Spring. We then conducted a detailed analysis of the framework's architecture and implementation details. During the implementation phase, we meticulously designed and executed a diverse range of test cases, experiments, and evaluations. These efforts provided valuable insights into the strengths and limitations of the framework, laying a solid foundation for future improvements. Some of them are listed below:

- **Different implementations of DHT**: Our current implementation has a replication of the DHT into each node. In our experiments, we found that the syncing does not have a significant effect in the performance of the system. Although, if the number of the nodes is vastly increased then due to the synchronization points the nodes will eventually be constantly locked. To further explain this, let's assume that we have 5.000.000 nodes and each sync request lasts 0.1 millisecond. The result is that during each sync round the node will be blocked for 5.000.000*0.1 milliseconds which is 500 seconds (~ 8.3 minutes). That means that the node will be still blocked from the previous syncing when the new syncing cycle is executed due to the interval time (5 minutes). To fix this issue we can either increase the sync interval which will affect our results or to reduce the load during synchronization. In order to decrease the load of the system, we can split the DHT into different nodes and as a result, only some nodes will contain the information about the nodes which contain a specific metric type. Additionally, we will have to implement a mechanism to perform requests to other nodes first in order to get the necessary information about the nodes, we should send the request for the data.
- **Usage of IPFS**: We would like to incorporate IPFS into our system, a cutting-edge technology that will revolutionize how we store and share data. By leveraging IPFS, we can enhance the resilience and efficiency of our platform.
- **Implementation of query aggregation mechanism**: Currently, we just fetch the data from the databases and display them in their original form. Although, we can implement a mechanism in order to aggregate the information to be able to extract knowledge from the existing data.

- **Implementation of security**: Currently, we don't have a mechanism in order to authenticate the user who sends the request. So everyone is able to send requests to every node. As our application can be used for sensitive information, it would be a good idea to implement a security layer to protect our system e.g. with the usage of a bearer token.
- **Usage of another database**: Other databases like RockDB and be used in order to create a new implementation to take advantage of their characteristics. Then we should compare the performance between the two implementations.
- **Usage of a database instead of DHT for the synchronization**: In order to avoid the synchronization between the nodes we can use either a central database like Redis or a distributed database like Citus data in order to save the abstract information about the content of the nodes.

In conclusion, this thesis significantly advances our understanding of edge computing, from theoretical concepts to practical applications. By introducing a new edge database tailored for distributed queries, it contributes substantially to the ongoing evolution of edge computing technologies. The insights gained and the proposed future directions offer promising avenues for further research and development, paving the way for more resilient and efficient edge computing solutions.

Appendix

Code Description

The development was conducted using Java 21 (openjdk 21.0.2), chosen for its stability and compatibility with modern Java applications. Apache Maven 3.9.6 facilitated project management by managing dependencies and building the project efficiently. Docker version 24.0.7 served as the containerization platform, providing a consistent and reproducible environment. These software versions were meticulously selected for their reliability and compatibility, making them ideal for smooth development and replication of the project.

Now let's move on to the description of the code. The description will take place file by file and the files will be ordered based on their packages.

SpringSqliteApplication.java

The first class which will be described is the SpringSqliteApplication class. The SpringSqliteApplication class serves as the entry point for the Spring Boot application. It initializes the application context, configures Spring Boot, and starts the application.

There are two important class annotations:

- **SpringBootApplication**: This annotation is known as a *meta-annotation*, and it combines @SpringBootConfiguration [25, 26], @EnableAutoConfiguration [25, 26] and @ComponentScan [24, 26]. It indicates that this class is the main configuration class for the application, enabling Spring Boot's autoconfiguration feature and component scanning. In more details, the mentioned annotations are used for:
    - **Spring Boot Auto-Configuration**: This annotation triggers Spring Boot's autoconfiguration mechanism, which automatically configures the application based on dependencies and conventions.
    - **Component Scanning**: With this annotation, Spring Boot scans for components, such as controllers, services, and repositories, within the same package and its sub-packages.
- **EnableScheduling**: This annotation enables Spring's scheduling capabilities, allowing the application to define scheduled tasks using @Scheduled annotations on methods.

Entities Package

ContainerEntity.java

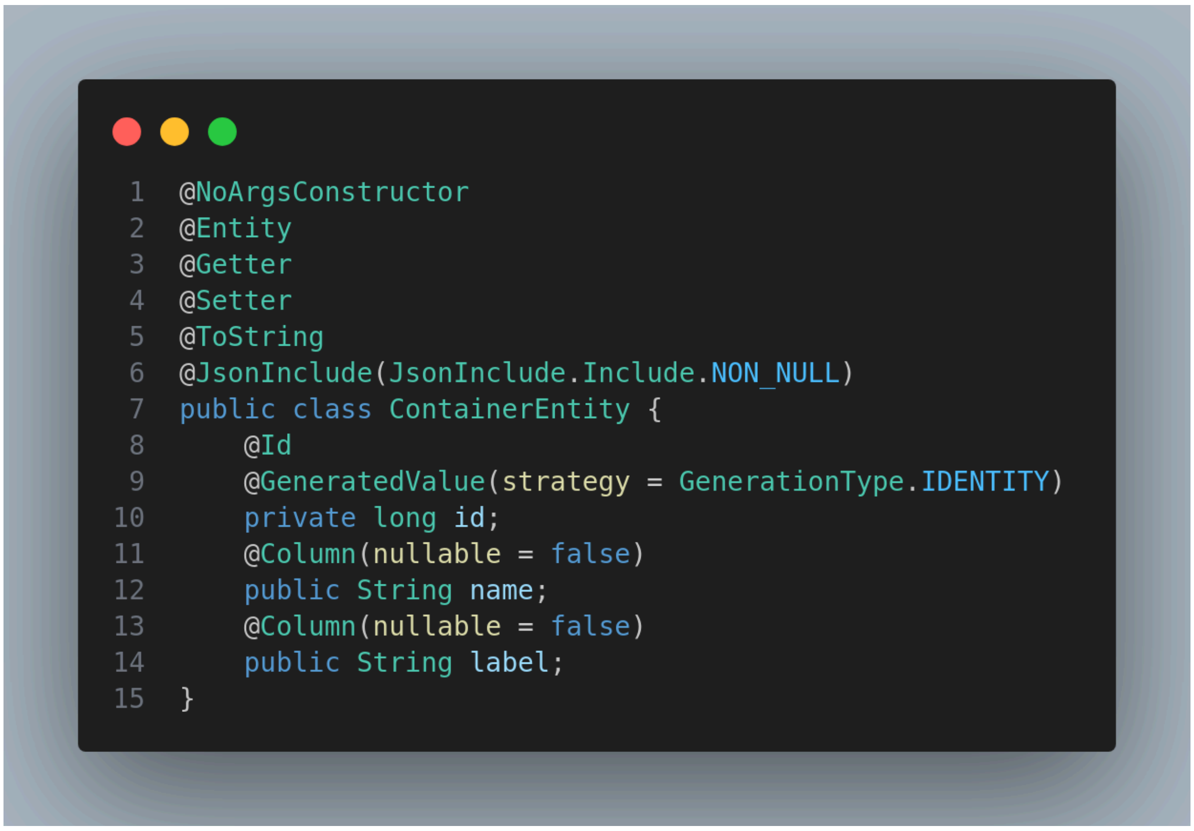
This class is used in order to create a table named Container in the database. This table will keep information about the containers which have put at least one metric in the metrics table.

It contains 3 fields:

- **id**: Id is the unique identifier of the container, and it is annotated with GeneratedValue annotation in order to auto create the value of this column based on the existing values
- **name**: It is the name of the container, and it is annotated with column annotation in order to ensure that this column won't have null value
- **label**: It is used to give a description to the container and like "name" field it is annotated with column annotation to ensure that this column won't have null value

Finally, the class is annotated with lombok annotations for getter, setter, noArgsContructor and toString in order to minimize the code and increase the readability of it.

For reference, the implementation of this class is visible below:



```java
@NoArgsConstructor
@Entity
@Getter
@Setter
@ToString
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ContainerEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(nullable = false)
    public String name;
    @Column(nullable = false)
    public String label;
}
```

34. Example of entity object configuration - Container Entity configuration

PodEntity.java

This class is used in order to create a table named Pod in the database. This table will keep information about the pods which have put at least one metric in the metrics table.

It contains 4 fields:

- **id**: Id is the unique identifier of the container, and it is annotated with GeneratedValue annotation in order to auto create the value of this column based on the existing values
- **name**: It is the name of the container, and it is annotated with column annotation in order to ensure that this column won't have null value
- **uuid**: It is the unique identifier of the pod inside the k8s system
- **namespace**: It is the k8s namespace where this pod was executed. By keeping this value, we are able to determine various information about the namespaces, for example which is the namespace which led to the heaviest load of the node

Finally, the class is annotated with lombok annotations for getter, setter, noArgsContructor and toString in order to minimize the code and increase the readability of it.

Enumerations Package

EntityTypeE.java

The EntityTypeE enum represents different types of metric types within the context of the application. It serves as a convenient and type-safe way to define and manage a fixed set of metric types. The available values are the following:

- **MEMORY_USAGE**: Represents the entity type related to memory usage metrics.
- **CPU_USAGE**: Represents the entity type related to CPU usage metrics.
- **DISK_USAGE**: Represents the entity type related to disk usage metrics.
- **GPU_USAGE**: Represents the entity type related to GPU usage metrics.

Utils Package

Utils.java

The Utils class provides constant variables in order to avoid their initialization in every class we want to use them. Also, it makes it easier to maintain the code, and it increases its readability. The following variables are present in this class:

- **LOGGER**: A static logger instance used for logging messages within the application. It is initialized using SLF4J LoggerFactory and provides logging capabilities to different components.
- **DISCOVERY_NODE_NAME**: A static constant representing the name of the discovery node retrieved from the system environment variable "DISCOVERY". It is used for detecting the “server” node if it fails to use the Kafka functionality.

- **HOSTNAME**: A static constant representing the hostname of the current node, retrieved from the system environment variable "HOSTNAME". It provides the hostname of the local machine, facilitating identification and communication with other nodes.

Controllers Package

ContainerController.java

The ContainerController class serves as a REST controller responsible for handling HTTP requests for adding or fetching containers within the SQLite application. Each container which is added through this controller is a potential producer of metrics. The controller is annotated with the following annotations:

- **RestController**: Indicates that this class is a REST controller, allowing it to handle incoming HTTP requests and return responses.
- **AllArgsConstructor**: Lombok annotation that generates a constructor with arguments for the class, injecting dependencies into the controller.
- **RequestMapping("/containers")**: Specifies the base URI path for mapping requests to this controller. That means that we are able to call the endpoint of this controller simply by sending a request to "http://localhost:{mapped-port}/containers". Each defined endpoint can have additional parts for its URI but it is always prefixed from the base URI.

This controller contains an autowired variable named containerService which is initialized from the constructor automatically. Also, it contains the following two methods:

- **addContainer**: Defines a new POST endpoint which is used to add a new container to the database. It expects a JSON payload representing a ContainerEntity object in the request body. Upon receiving a request, it invokes the addContainer method of the ContainerService to persist the container entity. Finally, it adds one extra part in the URI so the final endpoint url is "http://localhost/{port-mapping}/containers/add".
- **getContainers**: Defines a new GET endpoint for retrieving all containers from the database. It returns a list of ContainerEntity objects representing all containers stored in the database. This endpoint invokes the getAllContainers method of the ContainerService to fetch the container data. Finally, it adds one extra part in the URI so the final endpoint url is "http://localhost/{port-mapping}/containers/get".

PodController.java

The PodController class serves as a REST controller responsible for handling HTTP requests for adding or fetching pods within the SQLite application. Each pod, which is added through this controller, is a potential producer of metrics. The controller is annotated with the following annotations:

- **RestController**: Indicates that this class is a REST controller, allowing it to handle incoming HTTP requests and return responses.
- **AllArgsConstructor**: Lombok annotation that generates a constructor with arguments for the class, injecting dependencies into the controller.
- **RequestMapping("/pods")**: Specifies the base URI path for mapping requests to this controller. That means that we are able to call the endpoint of this controller simply by sending a request to "http://localhost:{mapped-port}/pods". Each defined endpoint can have additional parts for its URI but it is always prefixed from the base URI.

This controller contains an autowired variable named podService which is initialized from the constructor automatically. Also, it contains the following two methods:

- **addPod**: Defines a new POST endpoint which is used to add a new pod to the database. It expects a JSON payload representing a PodEntity object in the request body. Upon receiving a request, it invokes the addPod method of the PodService to persist the container entity. Finally, it adds one extra part in the URI so the final endpoint url is "http://localhost/{port-mapping}/pods/add".
- **getPods**: Defines a new GET endpoint for retrieving all pods from the database. It returns a list of PodEntity objects representing all pods stored in the database. This endpoint invokes the getAllPods method of the PodService to fetch the container data. Finally, it adds one extra part in the URI so the final endpoint url is "http://localhost/{port-mapping}/pods/get".

Repositories Package

ContainerRepository.java

The ContainerRepository interface, designed as part of the Spring Data JPA part of our application and serves as an abstraction layer for database interactions related to ContainerEntity objects within the SQLite application. It encapsulates methods for the retrieval of containers stored in the underlying database. This class contains the following 2 methods:

- **findAll()**: This method retrieves all ContainerEntity instances stored in the database and returns them as a list. It offers a straightforward approach to fetch all containers without any filtering criteria.
- **findByName()**: This method searches in the database for a container with a specific name. It returns Optional<ContainerEntity> which contains the target container if it is present in the database. We use this method in order to check if the target container of a new metric exists in the database or if we have to create it.

PodRepository.java

The PodRepository interface, designed as part of the Spring Data JPA part of our application and serves as an abstraction layer for database interactions related to PodEntity objects within the SQLite application. It encapsulates methods for the retrieval of containers stored in the underlying database. This class contains the following 2 methods:

- **findAll()**: This method retrieves all PodEntity instances stored in the database and returns them as a list. It offers a straightforward approach to fetch all pods without any filtering criteria.
- **findByName()**: This method searches in the database for a pod with a specific name. It returns Optional<PodEntity> which contains the target pod if it is present in the database. We use this method in order to check if the target pod of a new metric exists in the database or if we have to create it.

DHT Package

BaseRequest.java

The BaseRequest class serves as a foundational structure for transmitting requests within the distributed hash table (DHT) framework. It is designed to encapsulate essential information required for node synchronization and data distribution. The class implements the Serializable interface, enabling instances to be serialized and transmitted across the network. It contains the following three variables:

- **nodeName**: When a node sends a request to another node, it should denote its identity. This field stores the hostname of the sending node, allowing the receiver to handle the request appropriately.
- **currentNodes**: Each node maintains a list of known nodes in the network. During synchronization and other actions, nodes share this list to ensure mutual awareness. Further details on the utilization of this field will be discussed later.
- **dataMap**: This field maintains a map indicating which nodes contain specific types of metrics. This allows for targeted requests to nodes that can provide relevant results. More detailed explanations will be provided subsequently.

Additionally, this class is annotated with the following annotations:

- **AllArgsConstructor, Getter, ToString, and SuperBuilder**: Lombok annotations automate the generation of constructor methods, getters, toString() methods, and the builder pattern, respectively, enhancing code conciseness and readability.

- **JsonInclude(JsonInclude.Include.NON_ABSENT)**: This annotation specifies that only non-null values should be included during JSON serialization, reducing unnecessary data redundancy.

SyncRequest.java

The SyncRequest class extends the BaseRequest class, inheriting its structure and functionalities while adding a variable related to synchronization operations within the distributed hash table (DHT) framework. This variable, named deletedTypes, represents a set of entity types that have been deleted from the node's local storage. When all metrics of a specific type are removed from a node's database, it is essential to inform other nodes that the source node no longer contains information about this metric type. Consequently, other nodes will refrain from sending requests to this node for the specific metric type until new entries of this type are added to the node. Furthermore, this class manually implements its constructor to enable the invocation of the superclass constructor's implementation.

Additionally, this class is annotated with the following annotations:

- **Getter, ToString, and SuperBuilder**: Lombok annotations automate the generation of getters, toString() methods, and the builder pattern, respectively, enhancing code conciseness and readability.
- **JsonInclude(JsonInclude.Include.NON_ABSENT)**: This annotation specifies that only non-null values should be included during JSON serialization, reducing unnecessary data redundancy.

Services Package

ContainerService.java

The ContainerService class is annotated with @Service, indicating that it is a service component managed by Spring. Additionally, the @AllArgsConstructor annotation is used to generate a constructor with a parameter for each field in the class, enabling dependency injection of the ContainerRepository instance. It contains logic related to the management of container entities within the application. It serves as an intermediary between the controllers and the container repository, providing methods to perform create and read operations on container entities. This class has four method:

- **addContainer**: This method is responsible for adding a new container entity to the database. It takes a non-null container entity as input, saves it using the container repository's save method, and returns the saved entity.

- **getOrCreateContainer**: This method retrieves an existing container entity by name from the database if it exists. If the container does not exist, it creates a new container entity using the provided information and adds it to the database. This ensures that either an existing container is retrieved or a new one is created and returned.
- **getContainerByName**: This method retrieves a container entity by its name from the database, if available. It takes the name of the container as a non-null input parameter and returns an optional container entity, allowing for safe handling of potential null results.
- **getAllContainers**: This method retrieves all container entities stored in the database and returns them as a list. It does not require any input parameters and provides a comprehensive view of all containers present in the system.

PodService.java

The podService class is annotated with @Service, indicating that it is a service component managed by Spring. Additionally, the @AllArgsConstructor annotation is used to generate a constructor with a parameter for each field in the class, enabling dependency injection of the PodRepository instance. It contains logic related to the management of pod entities within the application. It serves as an intermediary between the controllers and the pod repository, providing methods to perform create and read operations on pod entities. This class has four method:

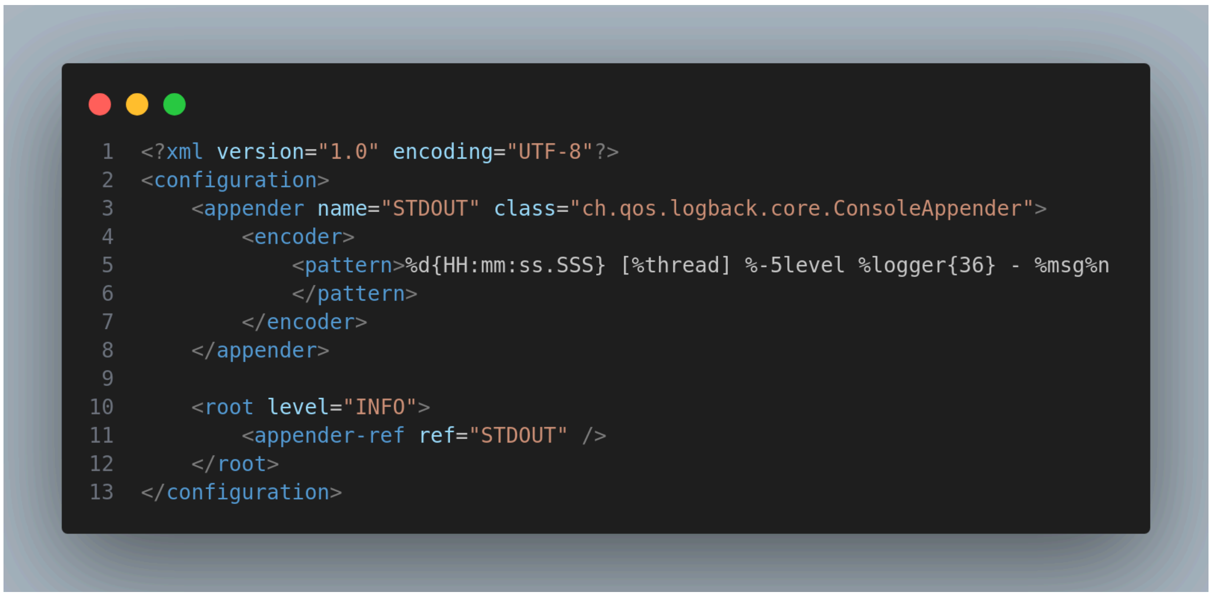
- **addPod**: This method is responsible for adding a new pod entity to the database. It takes a non-null pod entity as input, saves it using the pod repository's save method, and returns the saved entity.
- **getOrCreatePod**: This method retrieves an existing pod entity by name from the database if it exists. If the pod does not exist, it creates a new pod entity using the provided information and adds it to the database. This ensures that either an existing pod is retrieved or a new one is created and returned.
- **getPodByName**: This method retrieves a pod entity by its name from the database, if available. It takes the name of the pod as a non-null input parameter and returns an optional pod entity, allowing for safe handling of potential null results.
- **getAllPods**: This method retrieves all pod entities stored in the database and returns them as a list. It does not require any input parameters and provides a comprehensive view of all pods present in the system.

Other Files

logback.xml

The logback.xml file serves as a vital configuration tool for defining logging behavior within Java applications using the Logback logging framework [28]. We leverage this file to customize logging settings.

The following XML configuration specifies the formatting pattern for log messages and directs them to be output to the console appender named "STDOUT." The <pattern> element within the <encoder> tag defines the format of each log message. In our configuration, the %d, %thread, %level, %logger, and %msg placeholders are utilized to represent timestamp, thread name, log level, logger name, and log message content respectively. The %n represents a newline character. Furthermore, the <root> element configures the root logger's logging level to "INFO," indicating that log messages with an INFO level or higher will be output to the specified appender.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

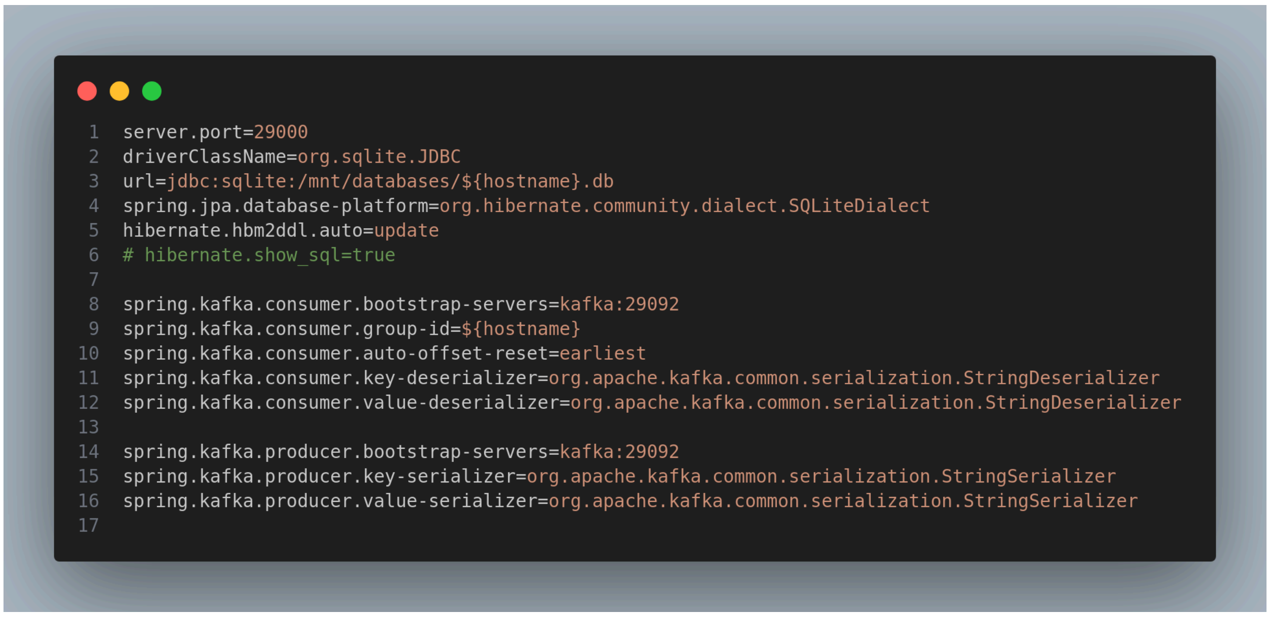
35. Example of logback configuration

application.properties

The application.properties file [29] serves as a central configuration file for defining various properties and settings within a Spring Boot application. We utilize this file to specify application-specific configurations, such as database connection details, Kafka integration settings, and other application properties.

The following application.properties file contains configuration properties related to the database connection and Kafka integration settings.

- **Database Configuration**:
  - **server.port**: Specifies the port number on which the server listens. By default, we use port 29000 as a fixed port. Later we will analyze how we use this port for reaching the running application.
  - **driverClassName**: Specifies the JDBC driver class for the SQLite database. In our case, the JDBC driver is from SQLite.
  - **url**: Specifies the JDBC URL for connecting to the SQLite database. The ${hostname} placeholder is used to dynamically generate the database URL based on the hostname.
  - **spring.jpa.database-platform**: Specifies the database dialect to be used by Spring Data JPA. In our case, it specifies the SQLite dialect.
  - **hibernate.hbm2ddl.auto**: Specifies the behavior of Hibernate schema generation. In this configuration, it is set to "update" to automatically update the database schema based on the entity mappings.
- **Kafka Integration Configuration**:
  - **Kafka Consumer Configuration**:
    - **spring.kafka.consumer.bootstrap-servers**: Specifies the list of Kafka brokers to connect to.
    - **spring.kafka.consumer.group-id**: Specifies the consumer group ID for Kafka consumers. It is dynamically set based on the hostname.
    - **spring.kafka.consumer.auto-offset-reset**: Specifies the offset reset behavior for Kafka consumers.
    - **spring.kafka.consumer.key-deserializer and spring.kafka.consumer.value-deserializer**: Specify the deserializers for Kafka key and value.
  - **Kafka Producer Configuration**:
    - **spring.kafka.producer.bootstrap-servers**: Specifies the list of Kafka brokers to connect to.
    - **spring.kafka.producer.key-serializer and spring.kafka.producer.value-serializer**: Specify the serializers for Kafka key and value.



```
server.port=29000
driverClassName=org.sqlite.JDBC
url=jdbc:sqlite:/mnt/databases/${hostname}.db
spring.jpa.database-platform=org.hibernate.community.dialect.SQLiteDialect
hibernate.hbm2ddl.auto=update
# hibernate.show_sql=true

spring.kafka.consumer.bootstrap-servers=kafka:29092
spring.kafka.consumer.group-id=${hostname}
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.producer.bootstrap-servers=kafka:29092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

```

36. Example of application.properties file

These configuration options ensure that the Spring Boot application is correctly connected to the SQLite database and integrates seamlessly with Kafka for messaging functionality. The dynamic properties, such as ${hostname}, enable flexibility and adaptability based on the runtime environment.

Docker

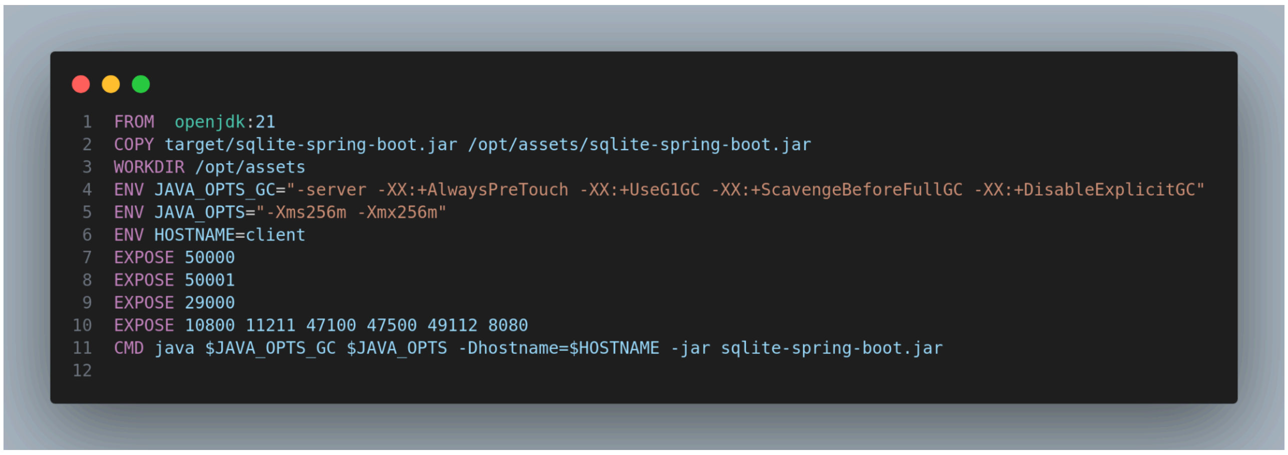
Dockerfile

The following Dockerfile [30] serves as a template for building Docker images, specifying the environment and instructions needed to create a containerized application. We use this Dockerfile to create the image which will be used in our nodes, which are a Docker container.

The provided Dockerfile contains instructions for building a Docker image for the application:

- **Base Image Selection**:
  - **FROM openjdk:21**: Specifies the base image to use for the Docker container, in our case the OpenJDK 21 image.
- **Application Setup**:
  - **COPY target/sqlite-spring-boot.jar /opt/assets/sqlite-spring-boot.jar**: Copies the application JAR file into the container's filesystem at /opt/assets.
  - **WORKDIR /opt/assets**: Sets the working directory within the container to /opt/assets, where the application JAR is located.
- **Environment Configuration**:

    - **ENV JAVA_OPTS_GC**: Sets the Java garbage collection (GC) options for runtime performance tuning.
    - **ENV JAVA_OPTS**: Sets the Java heap memory options for defining the initial and maximum heap sizes.
    - **ENV HOSTNAME**: Sets the hostname environment variable for the application. As described in application.properties file, we use this environment variable in order to dynamically set the database name as well as the group-id for our kafka consumer
- **Port Exposures**:
    - **EXPOSE**: Exposes various ports within the container for network communication. Ports 50000, 50001, 29000, 10800, 11211, 47100, 47500, 49112, and 8080 are exposed. In our current implementation only port 29000 is used which is the port where our spring application is running
- **Command Execution**:
    - **CMD**: Specifies the command to be executed when the container starts. It launches the Java application JAR with the configured JVM options and environment variables.



```
FROM  openjdk:21
COPY target/sqlite-spring-boot.jar /opt/assets/sqlite-spring-boot.jar
WORKDIR /opt/assets
ENV JAVA_OPTS_GC="-server -XX:+AlwaysPreTouch -XX:+UseG1GC -XX:+ScavengeBeforeFullGC -XX:+DisableExplicitGC"
ENV JAVA_OPTS="-Xms256m -Xmx256m"
ENV HOSTNAME=client
EXPOSE 50000
EXPOSE 50001
EXPOSE 29000
EXPOSE 10800 11211 47100 47500 49112 8080
CMD java $JAVA_OPTS_GC $JAVA_OPTS -Dhostname=$HOSTNAME -jar sqlite-spring-boot.jar
```

37. The Dockerfile which used for the generation of the image for the framework

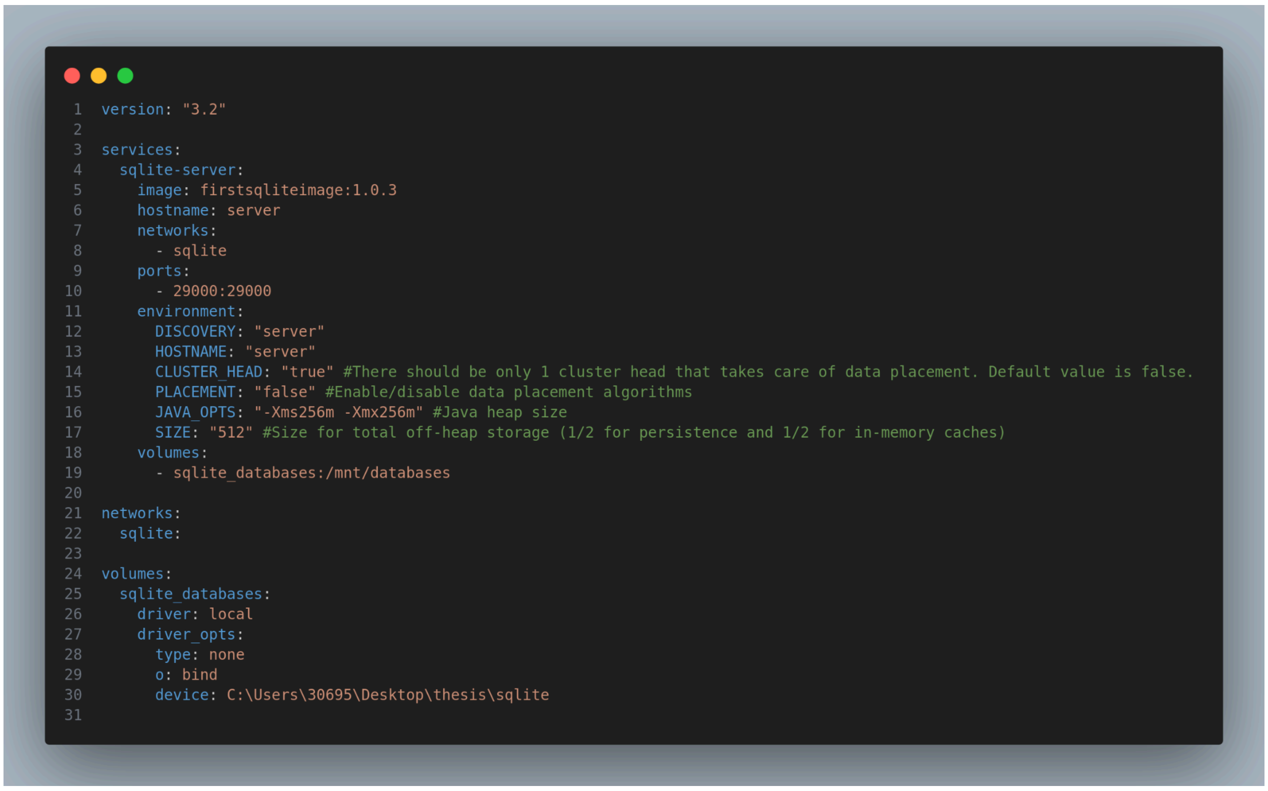
These configuration options ensure that the Docker image is built with the necessary dependencies and environment settings to run the Spring Boot application. The resulting Docker container is ready to execute the application with the specified runtime parameters, making it portable and easily deployable across different environments.

docker-compose.yml

Docker Compose files are used to define multi-container Docker applications. They allow us to specify the configuration for multiple services, including their dependencies, networking, volumes, and environment variables, in a single file.

The following Docker Compose file defines a service named "sqlite-server" and configures it with the following parameters:

- **Service Configuration**:
  - **image**: Specifies the Docker image to be used for the "sqlite-server" service, tagged as "edge-sqlite-db:1.0.0".
  - **hostname**: Sets the hostname of the container to "server".
  - **ports**: Maps port 29000 of the host machine to port 29000 of the container, enabling access to the service. By this way, we are able to reach this application through Postman in localhost:29000. For each new container, we map this port (29000) to a free port in order to be able to reach it from our desktop environment.
  - **environment**: Defines environment variables for the container, such as "DISCOVERY", "HOSTNAME", "CLUSTER_HEAD", "PLACEMENT", "JAVA_OPTS", and "SIZE".
  - **volumes**: Mounts the "sqlite_databases" volume to the container's "/mnt/databases" directory, allowing persistent storage for SQLite databases.
- **Network Configuration**:
  - **networks**: Creates a custom Docker network named "sqlite" for communication between containers. By this way, the nodes are able to communicate simply by sending request to {hostname}:29000.
- **Volume Configuration**:
  - **volumes**: Defines the "sqlite_databases" volume and specifies its driver type as "local". It also sets the volume's driver options to bind mount the host directory located at "C:\Users\30695\Desktop\thesis\sqlite" to the volume. If someone needs to execute this docker-compose file then this path should be changed to a path which exists in his/her machine.



```yaml
version: "3.2"

services:
  sqlite-server:
    image: firstsqliteimage:1.0.3
    hostname: server
    networks:
      - sqlite
    ports:
      - 29000:29000
    environment:
      DISCOVERY: "server"
      HOSTNAME: "server"
      CLUSTER_HEAD: "true" #There should be only 1 cluster head that takes care of data placement. Default value is false.
      PLACEMENT: "false" #Enable/disable data placement algorithms
      JAVA_OPTS: "-Xms256m -Xmx256m" #Java heap size
      SIZE: "512" #Size for total off-heap storage (1/2 for persistence and 1/2 for in-memory caches)
    volumes:
      - sqlite_databases:/mnt/databases

networks:
  sqlite:

volumes:
  sqlite_databases:
    driver: local
    driver_opts:
      type: none
      o: bind
      device: C:\Users\30695\Desktop\thesis\sqlite
```

38. Example of docker-compose file with services, networks, and volumes

docker-kafka-compose.yml

This Docker Compose file facilitates the deployment of a Zookeeper ensemble, Kafka cluster, and Kafdrop monitoring tool. It simplifies the process of setting up and managing these services by defining their configurations, dependencies, and network settings in a single file. We use this file to quickly deploy a local environment with Apache Kafka in order to use it for our application. Finally, it's worth mentioning that the created containers also utilize the sqlite network to enable access to Kafka via the kafka:29092 address from the containers running our application.

Troubleshooting

Should you encounter an issue during the startup of the Kafka service, such as receiving the error message "*kafka.common.InconsistentClusterIdException: The Cluster ID 7UXVApZBSF-dEyb1Ssy4Yw doesn't match stored clusterId Some(NJiX14e4RmWCiajSflaptw) in meta.properties. The broker is trying to join the wrong cluster. Configured zookeeper.connect may be wrong.*", it is advisable to ensure that all previous containers associated with Kafka have been properly deleted. This step is crucial for resolving any inconsistencies and ensuring the correct initialization of the Kafka service.

References

[1] Bigelow, S. J. (2021, December 8). What is edge computing? Everything you need to know. Data Center. https://www.techtarget.com/searchdatacenter/definition/edge-computing

[2] Knill, Z. (2023, July 31). So, you want to deploy on the edge? /Dev/Knill. https://zknill.io/posts/edge-database/

[3] What Is Edge Computing? | Microsoft Azure. (n.d.). https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-edge-computing/

[4] Valero, A., & Valero, A. (2023, August 22). Edge Computing: Empowering Real-Time Data Processing and Analysis. SUSE. https://www.suse.com/c/edge-computing-empowering-real-time-data-processing-and-analysis/#:~:text=Offline%20Operation%3A%20Edge%20devices%20can,in%20remote%20or%20disconnected%20environments.

[5]Kisters, S. (n.d.). What Are the Characteristics of Edge Computing? OriginStamp. https://originstamp.com/blog/what-are-the-characteristics-of-edge-computing/

[6] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., & Zhou, Y. (2017). Understanding the mirai botnet. In Proceedings of the 26th USENIX Security Symposium (pp. 1093-1110). (Proceedings of the 26th USENIX Security Symposium). USENIX Association.

[7] Multiple Vulnerabilities in Notepad++ Could Allow for Arbitrary Code Execution. (n.d.). CIS. https://www.cisecurity.org/advisory/multiple-vulnerabilities-in-notepad-could-allow-for-arbitrary-code-execution_2023-102

[8] SQLite Home Page. (n.d.). https://www.sqlite.org/index.html#:~:text=What%20Is%20SQLite%3F,database%20engine%20in%20the%20world.

[9] Paparrizos, J., Liu, C., Barbarioli, B., Hwang, J., Edian, I., Elmore, A.J., Franklin, M.J., & Krishnan, S. (2021). VergeDB: A Database for IoT Analytics on Edge Devices. Conference on Innovative Data Systems Research.

[10] Firebase Realtime Database. (n.d.). Firebase. https://firebase.google.com/docs/database

[11] Apache Cassandra | Apache Cassandra Documentation. (n.d.). Apache Cassandra. https://cassandra.apache.org/_/index.html

[12] Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/dynamodb/

[13] S. (2024, April 26). Unified AI Database - Azure Cosmos DB. Microsoft Learn. https://learn.microsoft.com/en-us/azure/cosmos-db/introduction

[14] EdgeDB | The next-gen database. (n.d.). https://www.edgedb.com/

[15] Yang, Y., Cao, Q., & Jiang, H. (2019). EdgeDB: An Efficient Time-Series Database for Edge Computing. IEEE Access, 7, 142295–142307. https://doi.org/10.1109/access.2019.2943876

[16] CockroachDB Pricing | Cockroach Labs. (n.d.). https://www.cockroachlabs.com/product/

[17] RocksDB | A persistent key-value store. (n.d.). RocksDB. https://rocksdb.org/

[18] Distributed Database - Apache Ignite&reg; (n.d.). https://ignite.apache.org/

[19] Edge Computing Architecture: A Practical Guide. (n.d.). https://www.run.ai/guides/edge-computing/edge-computing-architecture

[20] Mark Gamble, D. P. & S. M. (2024, May 1). An introduction to edge computing architectures. The Couchbase Blog. https://www.couchbase.com/blog/edge-computing-architecture-introduction/

[21]Understanding Edge Computing Solutions - Wipro. (n.d.). https://www.wipro.com/infrastructure/edge-computing-understanding-the-user-experience/

[22] What Is Edge Computing? 8 Examples and Architecture You Should Know. (2021, December 1). FSP TECHNOLOGY INC. https://www.fsp-group.com/en/knowledge-app-42.html

[23] Xiao, Y., Jia, Y., Liu, C., Cheng, X., Yu, J., & Lv, W. (2019, August). Edge Computing Security: State of the Art and Challenges. Proceedings of the IEEE, 107(8), 1608–1631. https://doi.org/10.1109/jproc.2019.2918437

[24] ComponentScan (Spring Framework 6.1.6 API). (n.d.). https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html

[25] Phillip Webb, D. S. (n.d.). Spring Boot Reference Documentation. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.configuration-classes

[26] Using Auto-configuration. Spring Boot Reference Documentation. (n.d.). https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.auto-configuration

[27] Bean. (n.d.). https://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch02s02.html

[28] Chapter 3: Logback Configuration. Chapter 3: Configuration. (n.d.). https://logback.qos.ch/manual/configuration.html

[29] Common application properties. (n.d.). https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

[30] Dockerfile reference. Docker Documentation. (2024, April 24). https://docs.docker.com/reference/dockerfile/

[31] GeeksforGeeks. (2023, July 28). Spring Boot - Crudrepository with example. https://www.geeksforgeeks.org/spring-boot-crudrepository-with-example/

[32] Distributed hash table. Synnada. (n.d.). https://www.synnada.ai/glossary/distributed-hash-table

[33] Suo, D. (n.d.). Peer-to-peer systems and distributed hash tables. COS 518: Advanced Computer Systems Lecture 15. https://www.cs.princeton.edu/courses/archive/spr17/cos518/docs/L15-dhts.pdf

[34] Distributed hash tables (dhts). Tutorialspoint. (n.d.). https://www.tutorialspoint.com/distributed-hash-tables-dhts#:~:text=Distributed%20databases%20%E2%88%92%20DHTs%20can%20be,retrieve%20large%20amounts%20of%20data.

[35] Distributed hash table. Hazelcast. (2023, November 16). https://hazelcast.com/glossary/distributed-hash-table/

[36] Hoek, J., Dai, Y., Lai, E., & Zhang, T. (n.d.). Peer to peer architecture. https://student.cs.uwaterloo.ca/~cs446/1171/Arch_Design_Activity/Peer2Peer.pdf

[37] Peer to Peer Architecture: An Easy Guide in 5 points (2021). UNext. (2022, July 6). https://u-next.com/blogs/product-management/peer-to-peer-architecture/

[38] What is peer-to-peer architecture (P2P architecture)? - definition from Techopedia. (n.d.). https://www.techopedia.com/definition/454/peer-to-peer-architecture-p2p-architecture

[39] Docker Overview. Docker Documentation. (2024a, April 17). https://docs.docker.com/get-started/overview/

[40] What is a container?. Docker. (2024, March 26). https://www.docker.com/resources/what-container/
[41] Apache kafka. Apache Kafka. (n.d.). https://kafka.apache.org/

[42] Cozens, B. (2022, March 23). Apache kafka: 10 essential terms and concepts explained. Red Hat - We make open source technologies for the enterprise. https://www.redhat.com/en/blog/apache-kafka-10-essential-terms-and-concepts-explained

[43] Introduction to spring framework. (n.d.). https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html

[44] Spring boot 3.2.5. Spring Boot. (n.d.). https://spring.io/projects/spring-boot

[45] JPA introduction - javatpoint. www.javatpoint.com. (n.d.). https://www.javatpoint.com/jpa-introduction

[46] DX Heroes. (2023, March 15). Apache kafka. Developer Experience Knowledge Base. https://developerexperience.io/articles/kafka

[47] Ultra Fast Distributed Cache. GhostDB. (n.d.). https://ghostdbcache.com/

[48] Eventql. (n.d.). Eventql/eventql: Distributed "massively parallel" SQL query engine. GitHub. https://github.com/eventql/eventql

[49] CITUS data: Distributed postgres. at any scale. Citus Data | Distributed Postgres. At any scale. (n.d.). https://www.citusdata.com/

[50] TiDB architecture. PingCAP Docs. (n.d.). https://docs.pingcap.com/tidb/stable/tidb-architecture

[51] What is edge computing?. IBM. (2023, June 20). https://www.ibm.com/topics/edge-computing