TU WIEN Informatics

Decentralized Epistemic-Based Communication Protocols for Volatile Edge Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Jakob Fahringer, BSc
Matrikelnummer 11830214

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Mag.a rer.soc.oec. Dr.in rer.soc.oec. Ivona Brandić

Wien, 8. Februar 2025

Jakob Fahringer	Ivona Brandić

TU WIEN Informatics

Decentralized Epistemic-Based Communication Protocols for Volatile Edge Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Jakob Fahringer, BSc
Registration Number 11830214

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Mag.a rer.soc.oec. Dr.in rer.soc.oec. Ivona Brandić

Vienna, February 8, 2025

Jakob Fahringer	Ivona Brandić

Erklärung zur Verfassung der Arbeit

Jakob Fahringer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 8. Februar 2025

Jakob Fahringer

Danksagung

Ich möchte mich herzlich bei Professorin Ivona Brandić für ihr wertvolles akademisches Mentoring bedanken. Ihre Unterstützung und ihre wissenschaftliche Expertise haben maßgeblich zur Entstehung und Weiterentwicklung dieser Arbeit beigetragen. Ein besonderer Dank gilt auch Shashikant Ilager, der mich in den ersten Phasen meiner akademischen Laufbahn begleitet und mir mit seinem Wissen und seiner Hilfsbereitschaft eine wertvolle Orientierung gegeben hat. Ebenso danke ich allen Freunden und Kollegen der HPC Research Group, die mich mit ihrem fachlichen Austausch und ihrer motivierenden Zusammenarbeit inspiriert haben.

Mein tiefster Dank gilt meiner Familie, die mich über all die Jahre hinweg unterstützt, mir Mut zugesprochen und mir den Freiraum gegeben hat, mich akademisch weiterzuentwickeln. Besonders dankbar bin ich auch meiner Freundin, die mir stets den Rücken gestärkt hat und mich in herausfordernden Zeiten motivierte. Ohne ihre Geduld, Unterstützung und Zuversicht wäre diese Arbeit nicht in dieser Form möglich gewesen. Ein großer Dank geht zudem an netidee, deren Förderung mir nicht nur finanzielle Erleichterung verschafft, sondern auch ermöglicht hat, mich voll und ganz auf meine Forschung zu konzentrieren.

Abschließend möchte ich mich bei allen Menschen bedanken, die mich während dieses Weges begleitet haben – sei es durch fachlichen Austausch, wertvolle Gespräche oder einfach durch ihr Vertrauen in mich und meine Arbeit.

Acknowledgements

I would like to sincerely thank Professor Ivona Brandić for her invaluable academic mentoring. Her support and scientific expertise have significantly contributed to the development and refinement of this work. A special thank you also goes to Shashikant Ilager, who guided me through the early stages of my academic journey, providing me with valuable orientation through his knowledge and willingness to help. I am also deeply grateful to all my friends and colleagues in the HPC Research Group, whose professional exchange and motivating collaboration have been a great source of inspiration.

My deepest gratitude goes to my family, who have supported me throughout the years, encouraged me, and given me the freedom to grow academically. I am especially thankful to my girlfriend, who has always stood by my side and motivated me through challenging times. Without her patience, support, and confidence in me, this work would not have been possible in its current form.

A great thank you also goes to netidee, whose funding has not only provided financial relief but has also enabled me to fully dedicate myself to my research.

Finally, I would like to express my gratitude to everyone who has accompanied me on this journey—whether through professional exchange, valuable conversations, or simply by believing in me and my work.

Kurzfassung

Die zunehmende Komplexität und Dynamik von Edge-Computing-Umgebungen stellt erhebliche Herausforderungen für traditionelle Monitoring-Systeme dar, die oft auf zentralisierte Architekturen angewiesen sind. Diese Ansätze führen zu Engpässen, begrenzen die Skalierbarkeit und schaffen Single-Points-of-Failure, wodurch sie für hochvolatile Edge-Netzwerke ungeeignet sind. Um diese Einschränkungen zu überwinden, stellt diese Arbeit DEMon vor, ein vollständig dezentrales Monitoring-Framework, das gossip-basierte Kommunikation und verteilte Datenabfrage nutzt, um skalierbares und robustes Monitoring zu gewährleisten. DEMon arbeitet ohne zentrale Koordination, wodurch die Knoten autonom Monitoring-Daten austauschen können, während der Rechen- und Netzwerkaufwand gering bleibt.

Eine umfassende experimentelle Evaluierung in einer containerisierten Testumgebung analysiert die Leistung des Frameworks über verschiedene Systemgrößen und Konfigurationen hinweg. Die Ergebnisse zeigen, dass DEMon Monitoring-Daten effizient selbst in großen Netzwerken verteilt. Durch die Feinabstimmung seiner Hyperparameter kann das System dynamisch zwischen Monitoring-Geschwindigkeit und Nachrichtenkomplexität abwägen, was eine Anpassung an unterschiedliche Einsatzumgebungen ermöglicht. Trotz seiner dezentralen Struktur bleibt die Datenabfrage aber zuverlässig, selbst wenn ein erheblicher Anteil der Knoten ausfällt. Dies wird durch Redundanzmechanismen sowie das Leaderless-Quorum-Consensus-Protokoll gewährleistet.

Die vergleichende Analyse mit FogMon2, einem hierarchischen Fog-basierten Monitoring-System als Referenz, verdeutlicht zusätzlich die Vorteile von DEMon. Während FogMon2 in kleineren Netzwerken eine geringere Nachrichtenkomplexität aufweist, skaliert DEMon deutlich besser und bleibt widerstandsfähig, ohne auf Leader-Knoten oder Aggregationsschichten angewiesen zu sein. Die Ergebnisse dieser Arbeit zeigen die Machbarkeit und Vorteile eines vollständig dezentralen Monitorings in Edge-Umgebungen, wodurch eine skalierbare und ausfallsichere Alternative zu bestehenden Lösungen geboten wird.

Abstract

The increasing complexity and dynamism of edge computing environments pose significant challenges for traditional monitoring systems, which often rely on centralized architectures. These approaches introduce bottlenecks, limit scalability, and create single points of failure, making them unsuitable for highly volatile edge networks. To address these limitations, this thesis presents DEMon, a fully decentralized monitoring framework that leverages gossip-based communication and distributed data retrieval to ensure scalable, and resilient monitoring. DEMon operates without a central coordinator, allowing nodes to autonomously exchange monitoring data while maintaining low computational and network overhead.

A comprehensive experimental evaluation in a containerized testbed examines the framework's performance across various system sizes and configurations. Results show that DEMon spreads monitoring data efficiently even in large networks. By fine-tuning its hyperparameters, the system can dynamically balance monitoring speed and message complexity, making it adaptable to different environments. Despite its decentralized nature, data retrieval remains reliable, even when a significant percentage of nodes fail, due to the system's redundancy mechanisms, as the Leaderless Quorum Consensus protocol.

The comparative analysis with FogMon2, a hierarchical fog-based monitoring system, as a baseline further highlights DEMon's advantages. While FogMon2 achieves lower message complexity in small networks, DEMon scales more effectively and maintains resilience without requiring leader nodes or aggregation layers. The findings of this thesis demonstrate the feasibility and advantages of fully decentralized monitoring in edge environments, offering a scalable and failure-resistant alternative to existing solutions.

Contents

CHAPTER 1

Introduction

The increasing adoption of edge computing has led to new challenges in monitoring highly dynamic and distributed environments. Traditional, centralized monitoring approaches struggle with fault tolerance, and volatility in such settings. Decentralized monitoring offers a promising alternative, leveraging peer-to-peer communication to ensure resilience. This thesis explores the feasibility of a fully decentralized monitoring system, focusing on its efficiency, resource consumption, and reliability in volatile edge environments.

1.1 Motivation

Monitoring is an essential component of distributed systems, ensuring stability, detecting failures, and optimizing performance. While centralized monitoring solutions have long been the standard, they struggle in environments where network conditions are unpredictable, and infrastructure is inherently volatile, such as in edge computing. Hierarchical fog-based solutions attempt to address this by introducing intermediary aggregation points, but they still suffer from bottlenecks and single points of failure. The need for a fully decentralized approach becomes evident when considering the scalability and resilience required in highly dynamic edge environments. Instead of relying on designated nodes or central aggregation servers, a full peer-to-peer model could distribute monitoring responsibility across all participants, improving fault tolerance and scalability. However, achieving this without introducing excessive overhead or compromising data accuracy remains a significant challenge. This thesis explores the feasibility of such an approach by improving and evaluating DEMon [IFDB22]—a fully peer-to-peer monitoring framework designed specifically for volatile edge environments. Using gossip-based communication and decentralized data retrieval, DEMon seeks to provide an efficient alternative to existing solutions. The research not only aims to validate the effectiveness of this model but also to identify its strengths and limitations in comparison to hierarchical alternatives.

1.2 Research Questions & Methods

This thesis investigates whether a fully decentralized monitoring system can efficiently operate in volatile edge environments without relying on hierarchical coordination. The research focuses on evaluating DEMon's scalability, resource efficiency, query performance, and comparative advantages over existing solutions. To achieve this, the following research questions(RQ) are examined:

RQ$_1$: *How quickly can DEMon discover all nodes in a distributed edge network, and what factors influence its convergence speed?* The gossip-based dissemination process is analyzed by measuring convergence time across different network sizes and configurations. By systematically adjusting gossip rate, gossip count, and system size, the study determines their impact on information propagation and system-wide convergence.

RQ$_2$: *What is the resource overhead of DEMon in terms of CPU, memory, and network usage?* DEMon's computational and network efficiency is quantified by tracking CPU and memory usage, as well as message complexity, during execution in a containerized edge testbed. Multiple test runs ensure consistency, and results are compared to other monitoring frameworks to contextualize its efficiency.

RQ$_3$: *How many messages are required for nodes to retrieve monitoring data, and how does the Leaderless Quorum Consensus (LQC) protocol influence query efficiency?* Query performance is evaluated by counting the number of gossip messages required for a node to retrieve monitoring data. The effectiveness of the LQC protocol in maintaining efficient and reliable data retrieval is tested under different network conditions, ensuring robustness even in failure scenarios.

RQ$_4$: *How does DEMon compare to existing decentralized monitoring systems like FogMon2 in terms of convergence speed and resource efficiency?* A direct comparative analysis is conducted using identical experimental conditions to evaluate differences in message complexity, convergence behavior, and resource consumption. This provides insights into DEMon's advantages and trade-offs relative to hierarchical alternatives.

To address these questions, an extensive experimental evaluation was conducted. A Docker-based testbed was used to deploy DEMon in a simulated edge environment, where nodes operated independently in isolated containers. Various hyperparameter settings were tested to assess their influence on convergence speed, resource consumption, and query efficiency. Additionally, FogMon2 was used as a baseline for comparison, allowing for a structured analysis of DEMon's performance against an established fog-based monitoring framework.

A preliminary version of this research was published in IEEE UCC 2022 [IFDB22],

introducing the core concepts and initial findings. Building upon that foundation, a more recent preprint [IFTB24] further extends the analysis. This thesis integrates and expands both works, providing a more comprehensive background, in-depth analyses, and refined methodologies to thoroughly address the research questions.

1.3 Expected Outcome

This thesis aims to demonstrate that a fully decentralized monitoring system can operate efficiently in volatile edge environments without relying on hierarchical coordination. The expected outcome is that DEMon will achieve scalable monitoring while maintaining low resource overhead. The experimental evaluation is expected to show that DEMon converges efficiently across different system sizes, with message complexity primarily influenced by gossip parameters rather than network topology. Additionally, it is anticipated that the information dissemination will ensure reliable data retrieval even under high failure rates.

A comparative analysis with FogMon2 should highlight that while DEMon eliminates hierarchical bottlenecks and improves scalability, it may introduce higher per-node resource consumption due to its tech stack. The findings are expected to provide insights into the trade-offs between decentralized and fog-based monitoring approaches, guiding future improvements in edge monitoring frameworks.

1.4 Overview

This thesis is structured to provide a comprehensive understanding of decentralized monitoring in edge environments. Chapter 2 introduces key concepts such as edge computing and decentralization, laying the foundation for the following discussions. Chapter 3 reviews existing monitoring solutions, including cloud-based tools and decentralized prototypes from research that focus on fog and edge monitoring. Chapter 4 outlines the scientific approach used in this work, presenting the research questions and methodology for addressing the identified challenges. Building on this, Chapter 5 expands on the methodology by detailing the system design of DEMon, explaining its theoretical foundations and functional architecture. Finally, Chapter 6 presents the evaluation results, visualizing the experimental findings and analyzing the impact of different parameters on system performance.

CHAPTER 2

Background

The following chapter is intended to serve as a knowledge base for readers of this thesis. It first explains the terms monitoring and monitoring systems in more detail before discussing edge-cloud monitoring. It also highlights the advantages and disadvantages of centralized and decentralized systems and how these affect monitoring.

2.1 Monitoring Systems

Monitoring, in general, describes the supervision of processes and data [Jö13]. Almost any type of observation of any system can be characterized as monitoring. The term has many uses, from medical applications [JW08] to the environment studies [KKH13]. However, in the context of this thesis, monitoring refers specifically to hardware and software. In this context, IT infrastructure or networks are considered, among other things.

2.1.1 Purpose and Objectives

One of the monitoring systems' main tasks is to monitor the system's health and/or the performance of individual participants [DC21]. To perform this task, some monitoring tools can identify anomalies [RBP11], [DS90] or performance bottlenecks [MH07], [NSV16] to ensure service operation or to improve them. Monitoring data can also be analyzed to predict future events in the overall system [GI01], [JVM13] and retrospectively identify sources of error [CLC22]. The partially analyzed data is then often accessible to system viewers, often through a user interface.

2.1.2 Components of Monitoring Systems

Depending on the tool, a monitoring system can be divided very differently. However, a typical system contains the following components:

- **Data Collection Agents**: These are programs closely linked to the hardware or software of the system. As the name suggests, their main task is collecting data on certain system parts. The collection is carried out cyclically or initiated by specific actions depending on the program [LR15]. The data includes information about specific resources, network activities, service status, etc. However, collectors should use as few resources as possible to keep the load on the monitored system as low as possible.
- **Data Aggregation and Storage**: Once the data has been collected, they are often summarized in larger packages. For example, the data of a data point can be grouped and subsequently saved. Aggregation guarantees that data from one source is stored and traceable in a standardized format. Depending on the aggregation method, this can also require less storage space.
- **Analysis and Visualization**: If humans monitor the system, the aggregated data are then analyzed by algorithms. The results of the analysis are processed graphically through a user interface, such as graphs, dashboards, or similar. This provides the user with a summary of the state of the system.

2.1.3 Types of Monitoring

As already mentioned, many types of monitoring can be categorized differently from varying perspectives. In the following, these categories refer to the focus areas of the monitoring system [Spl23]:

- **Infrastructure Monitoring**: Focuses on hardware resources such as CPU, memory, disk, and network utilization.
- **Application Monitoring**: Tracks the health and performance of applications, including error rates, request latency, and user experience metrics.
- **Network Monitoring**: Monitors network performance, packet flows, and bandwidth usage to detect issues such as congestion or link failures.
- **Security Monitoring**: Involves parsing and analyzing audit files to identify security threats.

2.1.4 Evolution and Trends

Traditional monitoring systems are designed to monitor centrally managed infrastructure [SGA+17]. Data is collected, aggregated, processed, and stored in one place. However, due to the rise of distributed systems and thus the edge cloud continuum, new approaches are being developed that are more compatible with the underlying distributed infrastructure. These modern systems are built to handle large-scale, geographically dispersed environments where latency, scalability, and fault tolerance are critical considerations [SGA+17].

In addition, the requirements for analyzing the recorded data are growing. There is a trend towards combining monitoring systems with machine learning to detect errors automatically [MWO+21]. Due to the conditions in an edge environment, lightweight solutions are also required.

2.2 The Edge-Cloud Continuum

The term edge cloud continuum describes the transition from distributed edge nodes through intermediate layers, such as fog devices, to central servers in the cloud [MPL+22]. This concept is becoming increasingly popular, as real-time applications require short communication paths and, therefore, low latencies [Kel24]. However, not all data can and should be sent directly to the cloud and processed there; aggregation and evaluation can occur in the immediate (fog) layer [LWW+22]. Computing units that operate in a distributed manner directly at the data source are called edge nodes [VWB+16]. Due to their low computing power and other rather fragile properties, edge nodes are usually not as reliable as larger cloud servers [VWB+16]. However, a new flexible form is emerging in this landscape that offers new possibilities but also challenges [VWB+16].

2.2.1 Definition and Core Concepts of the Edge

Edge computing decentralizes traditional cloud-based computing by moving processing power to the network's "edge" [VWB+16]. The "edge" refers to locations outside of centralized data centers, often near or on the devices where data is generated [VWB+16]. This paradigm shift is motivated by the growing volume of data from Internet of Things (IoT) devices [uA20], smart cities, autonomous systems, and other latency-sensitive applications.

Key principles of edge computing according to Khan et al. [KAH+19] among others are:

- **Proximity to Data Sources**: Placing the computation closer to the devices minimizes latency and enhances responsiveness.
- **Distributed Processing**: Tasks are distributed across edge nodes, ensuring efficient resource use and scalability of the system.
- **Context Awareness**: Edge systems can leverage local contexts, such as location and device-specific data, to improve decision-making and adaptability.

2.2.2 Benefits of Edge Computing

Edge computing provides several advantages over traditional cloud only architectures by enabling data processing closer to the source [KAH+19]. One significant benefit is reduced latency, as local computation eliminates the delays associated with transmitting data to and from centralized servers. Additionally, bandwidth consumption is optimized since only processed and relevant data is sent over the network, rather than raw information.

Another key advantage is improved reliability, as edge systems can function independently of cloud connectivity, ensuring continuous operation even in cases of network disruptions. Furthermore, edge computing enhances privacy and security by keeping data processing local, minimizing exposure to external threats and reducing the risks associated with transmitting sensitive information over public networks.

2.2.3 Challenges in Edge Computing

Despite its advantages, edge computing presents several challenges that must be addressed to ensure efficient and reliable operation. One of the primary concerns is resource constraint, as edge devices are typically small, mobile computing units with limited CPU, memory, and storage capacity. Applications must therefore be lightweight to function across different edge environments, and the decentralized nature of these systems increases the risk of failure if individual nodes become unreliable [VWB+16]. Additionally, intermittent connectivity poses a significant challenge [VWB+16], as unstable network connections can disrupt communication between nodes and networks, requiring robust fault tolerance and synchronization mechanisms to maintain system integrity. Scalability and management further complicate deployment, as handling a large number of distributed nodes with limited processing power demands efficient coordination strategies [VWB+16]. Service discovery and delivery also become more complex in dynamic edge environments where devices frequently join and leave the network, necessitating adaptive mechanisms to maintain seamless operation [VWB+16]. Moreover, enabling collaboration between heterogeneous edge computing systems introduces compatibility challenges, as different architectures, communication protocols, and resource limitations must be integrated effectively [VWB+16]. Lastly, deploying cost-efficient yet fault-tolerant models remains a critical challenge, as edge computing systems must balance resilience and affordability while operating under constrained conditions.

2.3 Decentral Systems

Decentralized systems and networks distribute decision making, data storage, and data processing in various independent parts [Bak08]. This process differs significantly from the centralized approach in which the tasks mentioned are processed at a central location. Decentralized systems are intended to operate independently of external influences. Functioning approaches to decentralized systems are characterized, above all, by robustness, resilience, and scalability [Bak08].

2.3.1 Characteristics of Decentralized Systems

Decentralized systems are characterized by several fundamental properties that enhance their robustness and adaptability. Control is distributed among participants rather than being managed by a central authority, reducing the risk of failures and attacks while increasing system resilience [VWB+16]. Fault tolerance is another key feature [VWB+16], as distributing data and operations across multiple nodes ensures continued

functionality even in the event of partial system failures. Additionally, decentralization can enable seamless scalability, allowing systems to expand by adding more nodes without significantly increasing architectural complexity [Gra86]. Autonomy further enhances system flexibility, as individual nodes can make independent decisions based on available information, ensuring efficient operation even in dynamic and unpredictable environments [Gra86].

2.3.2 Advantages and Challenges in Decentralised Systems

As mentioned decentralized systems offer several advantages, including enhanced resilience, improved privacy, and increased adaptability. By eliminating a central point of failure, these systems remain operational even if individual nodes fail, thereby increasing redundancy and security. Additionally, decentralization reduces the risk of data breaches associated with centralized storage, as sensitive information can be distributed or processed locally. Furthermore, decentralized systems can operate efficiently in dynamic environments, where conditions such as network availability or node configurations may change frequently.
However, decentralization also presents challenges. Coordinating actions among distributed nodes without a central authority can be complex, particularly in ensuring consistency and agreement. Communication between nodes can introduce latency and require additional overhead for synchronization. Moreover, while decentralization mitigates some security risks, it introduces others, such as vulnerabilities in consensus protocols. Efficient resource management across distributed nodes, especially in heterogeneous environments, remains a complex task.

2.4 Gossip-Based Protocols

Traditional communication models in distributed systems often rely on centralized coordination or hierarchical structures to efficiently distribute information [Kra18]. This approach is unsuitable for dynamic and highly volatile environments, such as edge computing, where nodes frequently join and leave the network, and network reliability is unpredictable [LSKT17].

To address these challenges, gossip-based protocols provide a decentralized, probabilistic data dissemination, and synchronization method [Bir07]. Inspired by epidemic models, these protocols ensure that information spreads efficiently across a network through random peer-to-peer interactions, making them highly fault-tolerant, and adaptable to unstable infrastructures.

Gossip protocols have been widely used in distributed databases, like Apache Cassandra [1] and monitoring frameworks [FGB21],[GFPB23] because they can provide low latency, efficiency, and resilience communication without introducing excessive network overhead.

[1] `https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archGossipAbout.html`, accessed 07-02-25

Their robust nature makes them particularly well suited for self-organizing monitoring applications in decentralized environments [DQA04], where data consistency and availability must be maintained without a single point of failure.

The following sections explore the core principles of gossip-based protocols, the different types of gossip mechanisms used in distributed systems, and the challenges they face when applied to edge computing and monitoring frameworks.

2.4.1 Principles of Gossip Protocols

Gossip-based protocols rely on randomized peer selection and iterative message exchanges to efficiently spread information across a network. Instead of broadcasting to all nodes, each participant selects a small subset of peers and shares updates, ensuring that data dissemination remains lightweight and scalable. Over multiple iterations, the information propagates exponentially, leading to eventual convergence without overwhelming the network compared to simple flooding [Bir07].

One of the core properties of gossip communication is its self-stabilizing nature [DQA04]. Nodes do not require global knowledge of the system; instead, they update their local state based on the information received from peers. Without external intervention, this decentralized approach allows the system to dynamically adapt to changes such as node failures, mobility, or network congestion [DQA04].

Another essential aspect is redundancy in information exchange, which improves reliability. Even if individual messages are lost due to network instability or temporary disconnections, the stochastic nature of gossiping ensures that the information will eventually reach all nodes through alternative paths, as shown in this thesis. This inherent fault tolerance makes gossip protocols particularly useful in volatile edge environments where infrastructure stability cannot be guaranteed [LSKT17].

In addition, gossip mechanisms can support different communication models according to application needs. Some variants prioritize low-latency updates by increasing peer selection frequency, while others optimize for minimal bandwidth usage by reducing redundant transmissions. By tuning these parameters, as discussed in the Evaluation-Chapter 6, gossip-based systems can balance efficiency, convergence speed, and resource consumption.

2.4.2 Types of Gossip Protocols

Gossip protocols can be classified according to how information is exchanged and their purpose in a distributed system [FKMR09]. Different variations optimize latency, network efficiency, and fault tolerance, making them adaptable to various environments.

- **Push Gossip**: In this model, nodes actively send updates to randomly selected peers at regular intervals [Jel11]. This method ensures fast initial dissemination, but can lead to redundant transmissions if many nodes already have the same data.

- **Pull Gossip**: Here, nodes request updates from peers instead of broadcasting information. This approach is practical when updates are infrequent, reducing unnecessary transmissions while ensuring that nodes can still retrieve missing data when needed [Jel11].
- **Push-Pull Gossip**: A combination of both push and pull mechanisms, where nodes simultaneously send and request information during each gossip cycle. This hybrid approach improves data synchronization speed while keeping overhead manageable [Jel11].
- **Selective Gossip**: Instead of randomly selecting peers, nodes use predefined topologies or ranking mechanisms to determine the most relevant recipients [UCR09]. This method is appropriate in systems where prioritization or geographic awareness can optimize dissemination [UCR09].
- **Aggregation Gossip**: Used for computing global statistics (e.g., average load, total resources available) in decentralized networks. The nodes exchange and update partial computations, ensuring that, the final result converges over multiple iterations [JMB05].

Each variant is tailored for specific trade-offs between speed, efficiency, and reliability, making gossip protocols versatile for monitoring, distributed storage, and decentralized control in dynamic environments.

2.4.3 Challenges in Gossip-Based Communication

Despite their scalability and resilience, gossip-based protocols face several challenges that impact efficiency, consistency, and resource consumption in distributed environments. Frequent message exchanges can lead to redundant data transmissions, increasing bandwidth consumption, especially in resource-constrained edge networks. Although gossip ensures eventual consistency, the speed at which all nodes synchronize depends on network size, topology, and message frequency, making real-time updates difficult in large-scale systems [Bir07]. If updates propagate unevenly, nodes may receive outdated or conflicting information, requiring additional mechanisms to resolve inconsistencies dynamically. In heterogeneous networks, nodes with intermittent connectivity or limited processing power may slow propagation or introduce data gaps [Bir07]. Optimizing factors like gossip frequency and peer selection is complex, as misconfiguration can waste resources or delay data synchronization.

2.5 Decentralized Information Dissemination

In distributed systems, efficient information dissemination is crucial to maintain up-to-date system states without relying on centralized coordination. Unlike traditional hierarchical or client-server models, decentralized dissemination methods ensure scalability, fault tolerance, and adaptability in dynamic environments.

These methods leverage peer-to-peer communication [MMP11], probabilistic message propagation [MMP11], and redundancy mechanisms [Bir07] to ensure that updates reach all nodes despite failures, delays, or network disruptions. The effectiveness of a dissemination strategy depends on factors such as latency, bandwidth efficiency, and resilience.

The following sections explore key dissemination approaches 2.5.1, retrieval mechanisms 2.5.2, and the role of trust and security 2.5.3 in decentralized environments.

2.5.1 Approaches

Decentralized information dissemination employs various strategies to ensure efficient data propagation without centralized control, each impacting latency, scalability, and network efficiency differently. Epidemic dissemination, inspired by the spread of biological viruses, involves nodes randomly interacting with peers to gradually propagate information throughout the network. While robust, this method can generate redundant messages, increasing bandwidth consumption [VvRB03]. Structured overlays, such as Distributed Hash Tables (DHTs), improve efficiency over pure gossip methods by directing queries to specific nodes instead of broadcasting information indiscriminately. Systems like Pastry [RD01] demonstrate this approach, offering efficient data retrieval in decentralized networks. Another strategy, cluster-based dissemination, organizes nodes into clusters where frequent intra-cluster communication is combined with controlled inter-cluster updates to balance local responsiveness and global efficiency. This method is exemplified in protocols like Spray and Wait [SPR05], used in delay-tolerant networks to optimize message delivery. Adaptive dissemination further enhances efficiency by dynamically adjusting update frequency and peer selection based on network conditions, reducing overhead while maintaining synchronization. Each of these strategies presents trade-offs in terms of efficiency, complexity, and resource consumption, necessitating careful selection based on the requirements of a given decentralized system.

2.5.2 Data Retrieval

In decentralized systems, efficient data retrieval is crucial to ensure fast and reliable access to distributed information without centralized indexing. Unlike traditional databases that query a single authoritative source, decentralized retrieval relies on various methods to locate data. Quorum-based retrieval involves sending queries to a subset of nodes and aggregating responses to ensure consistency and trustworthiness, a version of this approach is will be discussed in the Chapters 5 & 6 in the for of the Leaderless Quorum Consensus. Flood-based queries, as in [FZTS11], broadcast requests across the network, guaranteeing discovery but incurring high bandwidth usage. Structured lookups, such as those utilizing DHTs, store data deterministically, enabling efficient key-based queries with minimal overhead. Caching and replication strategies store frequently accessed data at multiple nodes, can reduce retrieval latency and improving resilience [ABGM90].

2.5.3 Trust and Security in Data Retrieval

Ensuring trust and security in decentralized data retrieval is challenging due to the absence of a central authority [BFL96] and the potential presence of malicious nodes. Without proper verification mechanisms, nodes may retrieve inconsistent, outdated, or deliberately manipulated data, compromising system reliability and decision-making processes [BFL96].

One fundamental technique for maintaining data integrity is cryptographic hashing [MXN+22], in which nodes generate and compare hash values to ensure that retrieved information has not been altered. Furthermore, quorum consensus [GPG19] mechanisms enhance trustworthiness by requiring a retrieval request to be validated only if most nodes provide matching responses, reducing the likelihood of accepting incorrect or manipulated data. To further strengthen trust, reputation-based systems track the reliability of individual nodes over time [ZWY+21], prioritizing responses from consistently accurate sources while limiting interactions with potentially malicious actors.

Beyond data integrity, encryption and access control mechanisms protect sensitive information by ensuring that only authorized nodes can decrypt and process specific datasets. This prevents unauthorized access while allowing for selective data sharing in multi-party environments. By combining these approaches, decentralized retrieval systems can mitigate risks such as Byzantine faults [CL+99], Sybil attacks [Dou02], and inconsistent data propagation, ensuring secure and verifiable information exchange in volatile edge environments.

CHAPTER 3

Related Work

Monitoring in distributed environments has been extensively studied across cloud, fog, and edge computing domains. Each paradigm presents unique challenges and requires different approaches to ensure efficient data collection, processing, and retrieval. Understanding existing monitoring solutions helps contextualize the need for decentralized approaches in volatile edge environments.

3.1 Cloud based Solutions

Cloud computing has long been the dominant model for hosting and managing applications. It offers centralized control, scalable resources, and monitoring services. Cloud monitoring solutions focus on resource utilization, performance tracking, and automated scaling and often leverage centralized architectures. However, when applied to highly dynamic and distributed edge environments, these systems face limitations where low-latency processing and decentralized data management are critical.

This section examines open-source and proprietary cloud monitoring solutions, analyzing their architectures, data collection methods, and capabilities for monitoring distributed edge devices.

3.1.1 Open-Source Solutions

Zabbix

Zabbix [1] follows a client-server model, where agents collect system metrics (CPU, memory, network) and push data to a centralized Zabbix server. It supports SNMP, IPMI, and JMX protocols for monitoring network devices and applications. Zabbix proxies allow

[1] `https://www.zabbix.com/`, accessed 07-02-25

Feature	Zabbix	Prometheus	AWS CloudWatch	Azure Monitor	Google Cloud Monitoring
Architecture	Client-server	Pull-based TSDB	Managed cloud service	Integrated telemetry	Fully managed cloud suite
Storage	SQL-based DB	Custom TSDB	AWS proprietary storage	Azure Metrics DB + Logs	Proprietary time-series DB
Edge Monitoring	Proxies for edge nodes	Exporters on IoT devices	CloudWatch Agent, Greengrass	Azure Monitor Agents, IoT Hub	Edge TPU, IoT Core
Query Language	SQL-like queries	PromQL	SQL-like Metrics Insights	KQL (Kusto Query Language)	MQL
Use Case	Enterprise infrastructure	Kubernetes, microservices	AWS-centric monitoring	Azure & hybrid cloud	Google Cloud workloads

Table 3.1: Comparison of Cloud-Based Monitoring Solutions

for scalability across large infrastructures, making edge device monitoring in distributed networks viable.

Prometheus

Prometheus [2] is a pull-based scraping metric from instrumented applications and services via HTTP endpoints. It uses a custom time series database (TSDB) and PromQL for querying. Designed for Kubernetes and microservices, it can monitor edge nodes by running exporters on lightweight IoT or fog computing devices, integrating with Grafana for visualization.

3.1.2 Proprietary Solutions

AWS CloudWatch

CloudWatch [3] collects metrics, logs, and events from AWS services and custom applications and stores data in a multi-tenant time series database. It supports CloudWatch Agent for edge devices, enabling custom monitoring of IoT sensors and edge servers. CloudWatch integrates with AWS Greengrass, optimizing edge monitoring in low-latency environments.

[2] `https://www.prometheus.io/`, accessed 07-02-25

[3] `https://aws.amazon.com/de/cloudwatch/`, accessed 07-02-25

Microsoft Azure Monitor

Azure Monitor [4] gathers telemetry data from Azure services, on-premise servers, and edge devices. It supports Azure Monitor Agents and Edge Monitoring Extensions for containerized edge workloads. Using Kusto Query Language (KQL), users can analyze edge performance, detect anomalies, and trigger actions in Azure IoT Hub.

Google Cloud Monitoring

Google Cloud Monitoringh [5] provides observability across Google Cloud, hybrid environments, and Kubernetes clusters. It collects metrics via APIs and Stackdriver agents and stores them in a proprietary time series database. Google's Edge TPU and IoT Core allow real-time monitoring of edge nodes, ensuring low-latency data retrieval and analytics.

3.2 Fog-Based Solutions

3.2.1 FMonE

FMonE is a decentralized, adaptive monitoring framework designed for fog and edge computing environments. Unlike traditional cloud-based solutions, it dynamically adapts to heterogeneous infrastructures and fluctuating network conditions, making it suitable for geo-distributed and resource-constrained systems. The framework is built around containerized monitoring agents, allowing for lightweight deployment and seamless integration with container orchestration tools like Marathon and Mesos.[BPM+18]

A key feature of FMonE is its orchestrated monitoring pipeline, which enables users to define custom workflows that specify which metrics to collect, how to process them, and where to store them. Implementing multi-layered data aggregation reduces bandwidth usage by filtering and processing data at different levels, from edge nodes to fog gateways and cloud servers. The system also incorporates self-adaptive capabilities, automatically detecting new nodes, adjusting the monitoring intensity based on workload fluctuations, and reallocating tasks in case of failures.[BPM+18]

Evaluations on a simulated fog testbed (Grid5000) demonstrated FMonE efficiency, with minimal performance overhead and automated failure recovery in seconds.[BPM+18]

3.2.2 FogMon

FogMon [FGB21] is a distributed, lightweight monitoring system explicitly designed for Fog computing infrastructures. It was developed to address the challenges of monitoring highly dynamic, resource-constrained, and heterogeneous environments in the Cloud-IoT continuum. Unlike traditional cloud monitoring solutions, FogMon is built to function

[4] `https://azure.microsoft.com/products/monitor`, accessed 07-02-25
[5] `https://cloud.google.com/monitoring/docs`

without centralized coordination, making it highly resilient to network failures, topology changes, and resource fluctuations [FGB21].

FogMon: A highly Decentralized Approach

The original version of FogMon [FGB21],[BFG19] introduced a peer-to-peer (P2P) self-organizing architecture, which ensures scalability and resilience in dynamic Fog environments. The system is based on a Leader-Follower model, where:

Follower nodes collect local hardware and network metrics such as CPU, memory, storage, latency, and bandwidth. Leader nodes aggregate data from multiple Followers and share it using gossiping protocols to ensure fast and reliable information propagation. To handle network failures and changing infrastructure, FogMon employs a latency-based clustering mechanism that dynamically adjusts which Followers report to which Leaders. This mechanism ensures that monitoring efficiency is maximized while bandwidth consumption is minimized. [FGB21]

Additionally, FogMon reduces overhead using a differential monitoring approach: instead of continuously transmitting raw data, nodes only send updates when a significant change in monitoring metrics is detected. This feature is crucial in resource-limited fog environments, where excessive communication can degrade system performance.[FGB21]

With FogMon2 [GFPB23] an updated version was released, which was also deployed

Adaptmon: A Self-Adaptive Extension of FogMon

AdaptiveMon [CTCM22] integrates a Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop to enable automatic system adjustments. Each monitoring node independently analyzes the collected data and applies predefined countermeasures based on the observed trends. This mechanism allows for:

Dynamic Adjustment of Sampling Rates: AdaptiveMon reduces the sampling frequency to conserve network bandwidth and power if a monitored indicator remains stable over time. In contrast, if fluctuations occur, it increases the frequency for improved accuracy. [CTCM22] Selective Monitoring of Metrics: When devices experience resource constraints (e.g., low battery), AdaptiveMon prioritizes critical metrics and disables non-essential ones, ensuring efficient resource utilization. [CTCM22]

3.3 Edge-Based Solutions

3.3.1 PyMon

PyMon [GK17] is a lightweight, container-based monitoring framework designed for resource-constrained edge environments, particularly Single Board Computers (SBCs) running Docker and Kubernetes. Unlike cloud and fog monitoring solutions, PyMon minimizes resource overhead while ensuring real-time CPU, memory, and network usage

monitoring in containerized services. Built on Monit, PyMon extends its capabilities to inspect Docker containers and transmits collected metrics to a central server for aggregation and visualization. Unlike cAdvisor and Prometheus, which are too resource-intensive for SBCs, PyMon uses on-device aggregation to reduce network bandwidth and processing overhead. [GK17] Using an ARM-based edge testbed, PyMon demonstrated low CPU and memory consumption while maintaining accurate monitoring data. It is categorized as an edge monitoring solution rather than fog-based, focusing on localized, direct monitoring rather than multilayered aggregation. [GK17]

3.3.2 DEMon: A Fully Decentralized Monitoring Solution

DEMon (Decentralized Edge Monitoring) is a fully decentralized self-adaptive monitoring framework designed specifically for highly volatile edge environments. Unlike all related work discussed in this thesis, which incorporates some form of centralization, DEMon is the only fully decentralized solution that eliminates central control, leader nodes, or aggregation servers, making it a key foundation for this research. [IFDB22]

DEMon uses a Gossip-based protocol for efficient and fault-tolerant information dissemination. This characteristic ensures monitoring data spreads dynamically across all nodes without creating network bottlenecks. It also introduces a Leaderless Quorum Consensus (LQC) protocol to address data consistency and trustworthiness, allowing nodes to retrieve accurate monitoring information without relying on centralized coordination. This approach significantly improves latency, fault tolerance, and resilience, making DEMon highly suitable for multi-party, resource-constrained edge environments.[IFDB22]

DEMon's containerized, lightweight architecture enables deployment on heterogeneous edge nodes, ensuring low overhead and high adaptability. Evaluations on large-scale Kubernetes-based testbeds demonstrate that the framework achieves rapid information convergence, resilient data retrieval, and efficient network load distribution even under extreme node churn.[IFDB22]

This thesis builds upon DEMon as a fundamental reference, as it is the only fully decentralized monitoring solution capable of operating without centralized storage, dedicated leader nodes, or external orchestration, making it a crucial step toward truly autonomous edge computing infrastructures.

CHAPTER 4

Methodology

This chapter details the experimental approach used to evaluate DEMon, including the research design, test environment, evaluation metrics, and validation strategies. It explains how the system is tested in a controlled Docker-based setup, focusing on convergence speed, resource overhead, and query efficiency. In contrast, Chapter 5 (System Design) describes the internal architecture and mechanisms of DEMon and the testbed.

4.1 Research Design

This work follows an experimental research methodology to evaluate the performance and efficiency of DEMon, a fully decentralized monitoring system designed for volatile edge environments. The main objective is to investigate how DEMon handles monitoring, data dissemination, and resource efficiency in distributed, large-scale edge networks. The study is structured around the following research questions:

RQ$_1$: ***How quickly can DEMon discover all nodes in a distributed edge network, and what factors influence its convergence speed?***
This question is addressed by analyzing the gossip-based information dissemination process, measuring information spread 6.3.1 and convergence time 6.3.3 across different network sizes and parameter settings. The experiments systematically adjust the gossip rate, gossip count, and system size to determine their impact.

RQ$_2$: ***What is the resource overhead of DEMon in terms of CPU, memory, and network usage?***
The research quantifies computational and network overhead by running DEMon in containerized edge environments, tracking message complexity 6.3.1 and CPU/memory usage 6.3.4. The overhead is measured over multiple runs, ensuring consistency and allowing for comparison with other monitoring frameworks.

RQ$_3$: ***How many messages are required for nodes to retrieve monitoring data, and how does the Leaderless Quorum Consensus (LQC) protocol influence query efficiency?***
To answer this, query efficiency is evaluated by counting the number of gossip messages required for a node to retrieve monitoring data. The role of the LQC protocol 5.3 in ensuring efficient, trustworthy data retrieval is assessed through controlled tests.

RQ$_4$: ***How does DEMon compare to existing decentralized monitoring systems like FogMon2 in terms of convergence speed and resource efficiency?***
This is addressed through a direct comparative evaluation with FogMon2, using identical test conditions to compare convergence behavior and memory overhead 6.3.5.

To systematically address these questions, we adopted a Docker-based testbed to emulate a large-scale, volatile edge environment in a controlled manner. This setup allows us to easily scale the number of edge nodes, introduce node failures, and repeat experiments under identical conditions. Each Docker container runs a DEMon agent and represents an independent edge node. Using containers on a 40-core Intel Xeon server with 128 GB RAM, we can emulate up to 300 concurrent edge nodes. This approach provides a realistic environment for evaluation: it reproduces key characteristics of edge deployments (distributed nodes with resource constraints) while ensuring experiments are reproducible and systematically variable. By using a single controlled server, we eliminate external network noise and guarantee that any performance differences are due to our system or parameters, not environmental inconsistency. This experimental setup is therefore appropriate for evaluating DEMon, as it offers scalability, realism, and control - all crucial for scientifically precise assessment of a decentralized monitoring system.

To ensure a fair and meaningful evaluation, we compare DEMon against a baseline monitoring system from the literature. We selected FogMon2 [GFPB23] as the baseline for comparison. FogMon2 is a state-of-the-art peer-to-peer monitoring solution specifically designed for Fog/Edge environments. While other monitoring frameworks exist (e.g., PyMon [GK17] or FMonE [BPM$^+$18]), FogMon2 was deemed most suitable because its design and goals closely align with ours. In particular, FogMon2 employs a decentralized approach (a hierarchical two-layer P2P architecture) that partially resembles gossip-based dissemination, making it an apt benchmark against our fully decentralized, gossip-driven system. FogMon2 organizes nodes into Leader and Follower roles. Follower nodes report local metrics to leader nodes, and leaders exchange information among themselves in a gossip-like peer-to-peer fashion. This architecture contrasts with DEMon's flat gossip network (as every node is equal and communicates via gossip), allowing us to study the impact of hierarchy vs. full decentralization. Furthermore, FogMon2 is a recent, robust solution shown to handle edge volatility and is actively reported in literature, which legitimizes it as a strong baseline. By tuning FogMon2's configuration (especially the

number of leader nodes) as recommended by its authors, we ensured FogMon2 operates at optimal settings during comparisons.

Another key design choice in our methodology is the use of a gossip based approach for data dissemination in DEMon. We specifically leverage a gossip protocol because it aligns well with the challenges of highly volatile edge environments. Gossip (or epidemic) communication offers several advantages vital for our context: it has no single point of failure, distributes load uniformly, and scales naturally as nodes join or leave [Bir07]. Each node periodically selects a few random peers to exchange state information, which leads to an exponential spread of updates throughout the network. This stochastic peer selection ensures no particular network link or node is overwhelmed, avoiding bottlenecks. In a volatile environment where nodes may fail or disconnect unexpectedly, gossip's redundancy and randomness mean the system can continue operating: information eventually reaches all nodes despite failures and transient outages do not disrupt the overall monitoring process. New nodes can be integrated by contacting any one existing node in the network and then propagating their presence via gossip. These properties are essential for robustness: DEMon's decentralized gossip ensures that even if multiple nodes fail, the monitoring data is not lost since it's been disseminated to many others. Additionally, our gossip implementation is optimized to avoid unnecessary data transmission – nodes first exchange metadata and only transfer full data if the other peer is missing an update. This design minimizes bandwidth usage while still achieving consistency.

4.1.1 Experimental Procedure and Variables

We conducted a series of experiments to evaluate DEMon's performance and answer our research questions. Each experiment is structured to isolate the impact of specific variables while controlling others, following a scientific experimental design as stated. We identified several key parameters (independent variables) that influence DEMon's behavior and configured test runs to vary these systematically:

- System size: The number of nodes in the network. We scaled n from 50 up to 300 (in increments of 50) to assess scalability. This tests how our system performs as the edge environment grows, ensuring that results are not limited to small-scale scenarios.

- Gossip frequency: The interval (in seconds) between gossip rounds (=`gossip_rate`). We experimented with rates of 1, 5, 10, 15, and 20 seconds. A lower `gossip_rate` means nodes communicate more frequently. Varying this parameter allows us to observe the trade-off between convergence speed and network overhead.

- Gossip fan-out: The number of peer nodes each node contacts per gossip round (=`gossip_count`). We tried values of 2, 3, and 4. Higher `gossip_count` can

accelerate information spread but may increase message load; this parameter helps to find the optimal dissemination degree.

- Failure rate: The percentage of nodes that are intentionally failed (disconnected) during an experiment. We tested failure rates from 0% (no failure) up to 90%, in increments of 10%. This extreme range probes DEMon's resilience: even at 90% node loss, the system's ability to recover or retain information is evaluated.
- Baseline leader configuration: For FogMon2, we ran separate experiments with varying numbers of leader nodes. This was done to observe FogMon2's performance under different recommended configurations and to identify its best-case scenario for fair comparison with DEMon.

For each experimental run, we follow a consistent procedure: All nodes (containers) start with the DEMon agent (or FogMon2 agent for baseline tests), begin collecting monitoring information and gossip this data. We allow the system to run until a convergence condition is met or a fixed time limit is reached. We define convergence in DEMon as the point when every node in the network has learned about all other active nodes and their state. In FogMon2, convergence is reached when all leader nodes have exchanged information such that every leader knows at least one state of each follower. This convergence criterion is crucial for measuring certain metrics (like total messages or time to convergence) consistently across both systems.
During each run, we collect detailed logs and metrics. DEMon nodes are instrumented to count gossip messages sent and received, track state repository size, and record timestamps of events (e.g., when a new node state is first received). Similarly, we instrumented FogMon2's code (via minor modifications) to gather the number of messages exchanged among leaders and followers until convergence, as well as resource usage statistics. All experiments were repeated multiple times to ensure reliability: We report average values to smooth out any fluctuations. We also reset the environment between runs to avoid any residual state (e.g., restart the agents/containers) so that each trial starts from a clean slate.
Throughout all tests, we carefully control variables and the effect of one parameter (e.g., `gossip_rate`), as we keep all others at a default or baseline value to attribute differences in outcomes to that parameter alone. This one-factor-at-a-time approach ensures our analysis of each variable is independent. The combination of systematic variation, repeated trials, and controlled conditions contributes to an exact experimental methodology. In the next subsections, we detail the evaluation metrics we used, explaining how each is measured, why it was chosen, and how it relates to our research questions and hypotheses.

4.2 Evaluation Metrics

To thoroughly evaluate DEMon and compare it with FogMon2, we considered multiple performance metrics. Each metric targets a specific aspect of system behaviour relevant

to the proposed research questions. Below, we introduce each metric, describe how it is measured in our experiments, and justify its relevance.

4.2.1 Convergence Efficiency

Convergence efficiency captures how quickly and with how much communication effort the monitoring information spreads to all nodes. This metric is critical for addressing RQ1, which asks about DEMon's ability to rapidly disseminate monitoring data across a dynamic network. We evaluate convergence efficiency in two complementary ways:

- Convergence time: The elapsed time (in seconds or number of gossip rounds) for the system to reach convergence. This indicates the speed at which a consistent global view is achieved. We measure this by timestamping when the first node is starting till each node first becomes aware of all other nodes. The largest of these times (across all nodes) is taken as the convergence time for that run. We repeat this for different gossip parameters. This measurement is straightforward in our logs because once the framework knows the number of active nodes beforehand. Convergence time is important because in edge environments, faster information propagation means the monitoring system can react to changes (like new nodes or failures) more quickly.

- Communication overhead: The total number of gossip messages exchanged in the system up to the point of convergence. We instrumented every DEMon agent to count each message sent and received, summing across all nodes. For FogMon2, we similarly count the total messages passed among followers and leaders until its convergence criterion. This metric reflects the bandwidth usage and scalability of the protocol – fewer messages for convergence implies a more efficient dissemination. Monitoring solutions should minimize communication overhead to avoid overloading the network, especially in bandwidth-constrained edge scenarios. By comparing DEMon and FogMon2 on this metric, the work shows how the fully decentralized gossip impacts network load relative to the hierarchical approach. Prior work suggests that gossip protocols can spread information exponentially while keeping load balanced [Bir07], so we expect DEMon to require a comparable number of messages to FogMon2, especially as the system scales up.

Convergence speed and overhead directly determine the timeliness and efficiency of the monitoring system. A low convergence time ensures that all nodes have up-to-date information rapidly, which is vital for any real-time monitoring or automated decision-making at the edge. Meanwhile, low communication overhead indicates the solution is scalable and won't overwhelm the network or nodes as the number of devices grows. Together, these sub-metrics tell us how well DEMon meets the demands of fast, efficient data dissemination (RQ1) in large, volatile networks.

4.2.2 Resource Utilization

Resource utilization metrics assess how much load the monitoring system imposes on each node in terms of CPU and memory usage. This addresses RQ2, concerning DEMon's suitability for resource-constrained edge devices. Even a fast and fully-informed monitoring system would be impractical if it consumes excessive resources on edge nodes, which often have limited CPU power or memory. We measure CPU usage by monitoring the utilization percentage of the DEMon agent process on each container during experiments. Using Docker, we sample CPU load at regular intervals and compute an average usage per node. We similarly track memory usage, focusing on two aspects: the in-memory state repository size and the overall memory footprint of the agent process. The state repository holds monitoring data from across the network; as the system converges, this grows to contain records for all nodes. By measuring its size at convergence for different system scales, we quantify how much memory DEMon needs to store global states. We also observe memory growth over time (it increases as new data arrives, then stabilizes after convergence). For network overhead, although partly reflected in message counts (as discussed above), we also consider bandwidth usage. We measure the volume of data transmitted per node in each gossip round (in bytes) and track how it accumulates until convergence. This gives a more fine-grained view of network load than just counting messages, since messages could vary in size. In our experiments, we found that bandwidth usage is high during initial gossip rounds (when many new updates are flowing), but drops to lower levels after convergence, indicating that DEMon does not continuously flood the network. For comparison, we also measured FogMon2's resource usage under its optimal configuration on our hardware. FogMon2's CPU and memory consumption were recorded when running 80 nodes (the maximum FogMon2 could handle on our server). This provides a baseline to ensure DEMon's resource overhead is competitive. By measuring CPU, memory, and storage requirements, we verify that DEMon can run on edge nodes without straining their limited resources. An effective edge monitoring solution must "do more with less," so demonstrating low resource utilization is key to its practicality (a direct focus of RQ2). These metrics are also tied to system scalability: if CPU or memory usage grows too large with number of nodes, that would indicate a scalability bottleneck.

4.2.3 Resilience and Query Performance

To address the question of fault tolerance (RQ3), we evaluate how DEMon performs when a significant portion of nodes fail. The primary metric here is query success rate, which is defined by how many queries to different nodes are needed to get the newest monitoring data from a specific node. In a decentralized monitoring system, even if many nodes go offline, the remaining nodes should ideally still hold the departed nodes' last known data and be able to serve queries about them. We test this by measuring how reliably and quickly the system can retrieve monitoring information when up to X% of nodes have failed. Our experimental procedure for this metric is as follows: after an initial convergence with all nodes, we randomly disconnect a fraction of nodes (according

to the failure rate being tested). We then issue monitoring data queries from a subset of surviving nodes, where each query asks for the latest data of a randomly selected target node. Some of these target nodes may be among the failed ones, and others may be still active. We use DEMon's query API (via the DEMon Query Client) to fetch the data, which triggers the Leaderless Quorum Consensus retrieval mechanism in the proposed system. We performed 100 queries for each failure scenario. We log whether each query was answered successfully (= the quorum consensus was met) and how many messages were involved in retrieving the data. Impressively, even with extreme failure rates (up to 90% of nodes offline), DEMon continued to return the requested node's data in every query, with no failures. This is because the gossip dissemination ensures that every node's state is replicated across many others. Therefore, even if the origin node is down, its information lives on in the network. The number of messages required to satisfy a query naturally increases with more failures (since fewer nodes have the data, the query may need to ask more peers), but even in the worst case (90% failures) the largest observed query took only 148 messages to retrieve the data. On average, queries under failure conditions required about 10 messages, and in the best cases as few as 3 (which displays the minimum number of queries as the quorum number in our LQC protocol). DEMon's design choice of full decentralization with replication inherently provides resilience. We quantitatively demonstrate this via the above query success and message-count metrics under stress.
This resilience metric is crucial for validating that DEMon meets its goal of operating reliably in volatile conditions. Edge environments are prone to unexpected node outages. The proposed metrics directly test the system's core promise: can it still serve monitoring-data when a large portion of the network is gone? By measuring query success and required effort after failures, we connect to RQ3, which concerns reliability and fault-tolerance. A high success rate with acceptable latency means the system effectively tolerates failures. This metric thus provides evidence of DEMon's robustness.

4.2.4 Age of Information

Finally we examine the freshness of the monitoring data across the system using the metric of Age of Information (AoI). AoI is defined as the time elapsed since the last update of a particular piece of information. In our context, for each node X we can define the age of information at node Y as how old the state of X is in Y's repository (i.e., the difference between the current time and the timestamp of X's last gossiped state that Y has). This metric was chosen to address the question: Even after convergence, how up-to-date is the information that each node holds about others? This relates to RQ1's concern with timely information exchange and also to practical needs of monitoring, as outdated data can mislead further decision. We measure AoI during our experiments by tagging each state update with a timestamp and then periodically calculating the age of every node's knowledge of every other node. Our interest is in how AoI evolves over time, especially after initial convergence. In a stable state where nodes still gossip periodically, we expect that if no new changes occur, AoI might increase linearly (information getting older with ongoing time) until the next gossip exchange refreshes it. If nodes are continuously

updating (e.g., their resource usage metrics change), those changes will propagate, and AoI will reflect the update frequency. We specifically analyzed AoI under a high-frequency gossip setting (-> `gossip_rate` = 1) to see the best-case freshness, as well as under slower gossip to note any staleness. We found that DEMon can keep the AoI linearly for all critical metrics. Age of Information directly reflects the usefulness of the monitoring data at any node. Even if a system converges quickly and is robust, if it provides only stale information, its value can be limited by old data. By quantifying AoI, we address the quality of monitoring service as data freshness. This is especially relevant for real-time monitoring applications. AoI is thus tied to our research objectives on timely data dissemination. It also helps tune the system: for example, if AoI is too high, one might decrease the `gossip_rate` or/and increase `gossip_count` to improve freshness. In our methodology, including AoI ensures our evaluation is comprehensive: not just measuring how fast and efficiently data spreads (previous metrics), but also how current that data remains during operation.

CHAPTER 5

System Design

This chapter provides a detailed overview of DEMon's architecture, implementation, and operational workflow. It describes the system's core components, including data dissemination 5.1, retrieval mechanisms 5.2, and the structure of the State Repository (SR) 2. Additionally, it explains how DEMon ensures efficient monitoring while maintaining a fully decentralized architecture.

A key focus is given to information propagation using a gossip-based approach, the Leaderless Quorum Consensus protocol 5.3 for data retrieval, and the technical implementation using lightweight containerized deployment. Furthermore, the transition from a Kubernetes-based testbed to a Docker-based deployment is discussed, highlighting the improvements in scalability, efficiency, and decentralization.

The final sections outline the system's API, dependencies, and execution requirements, followed by an overall workflow description to illustrate the end-to-end monitoring process within DEMon.

5.1 System Architecture

DEMon is designed as a fully decentralized monitoring system for highly dynamic edge environments. Unlike fog-based solutions that rely on centralized aggregation nodes, DEMon ensures that each node independently manages monitoring, storage, and data retrieval without hierarchy. This architecture enhances fault tolerance, scalability, and adaptability, making it ideal for edge networks where devices frequently join and leave.

At the core of DEMon's design are four key components, as seen in Figure 5.1: the State Repository (SR), the Information Dissemination Controller (IDC), the Information Retrieval Controller (IRC), and the DEMon Query Client (DQC). These modules work together to facilitate continuous monitoring, decentralized data propagation, and efficient

query resolution, ensuring that monitoring data is always accessible without relying on dedicated leader nodes.

Each DEMon instance operates independently, continuously collecting local system metrics such as CPU usage, memory consumption, and network activity. This data is stored in the State Repository (SR), a lightweight local storage module that maintains a historical record of monitoring information. Instead of forwarding data to a central aggregator, each node participates in a peer-to-peer (P2P) gossip protocol, where the Information Dissemination Controller (IDC) ensures that monitoring updates are efficiently propagated across the network. The decentralized nature of this protocol prevents bottlenecks and ensures resilient data dissemination even in network failures.

The Information Retrieval Controller (IRC) processes monitoring queries within the distributed network to enable real-time data retrieval. Demon uses a Leaderless Quorum Consensus (LQC) protocol to verify and fetch monitoring data directly from peers, ensuring that retrieved data is accurate and consistent without requiring a centralized lookup service. External applications or services can access this monitoring data through the DEMon Query Client (DQC), which provides a structured interface for fully decentralised querying system states.

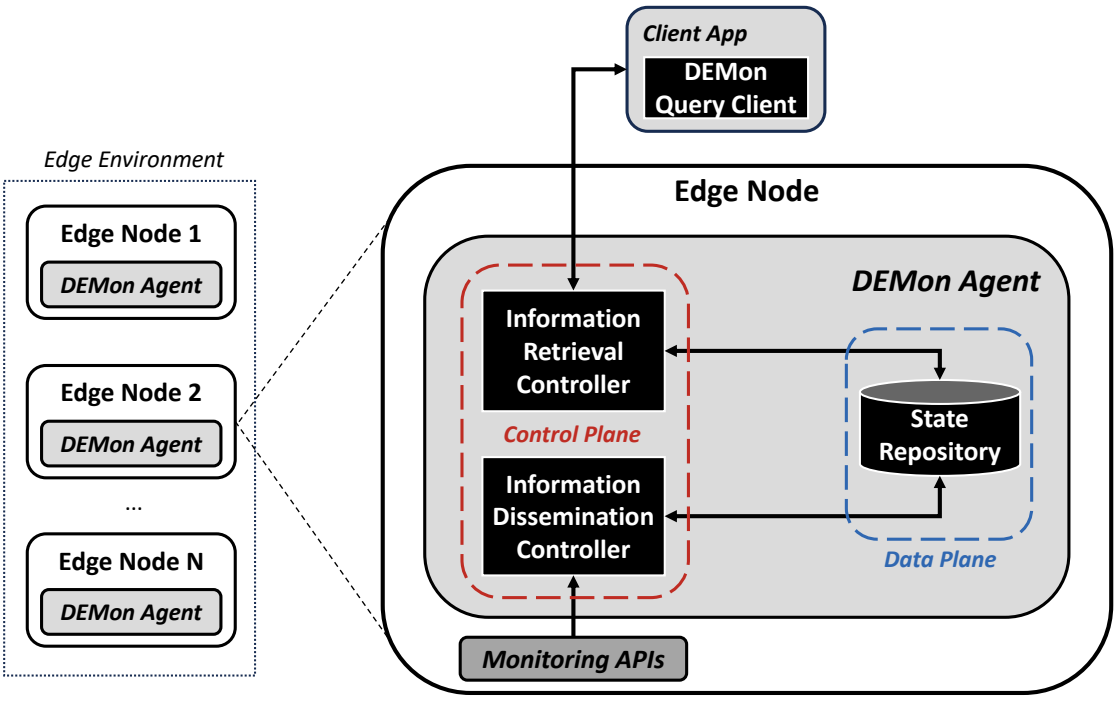


Figure 5.1: Schematic overview of the DEMon components [IFTB24]

This architecture makes DEMon highly scalable and resilient, as there is no single point of failure. Even if multiple nodes disconnect or experience failures, the remaining nodes continue operating autonomously, maintaining an accurate and up-to-date monitoring

system. Containerized deployments further enhance portability, allowing DEMon to deploy across heterogeneous edge infrastructures easily.

5.2 Gossip Protocol Design

DEMon utilizes a gossip-based information dissemination protocol to ensure monitoring data is propagated efficiently throughout a fully decentralized edge network. This approach enables scalable and fault-tolerant data sharing, ensuring that all nodes maintain an up-to-date view of the system state without requiring central coordination. The gossip mechanism in DEMon follows a probabilistic, peer-to-peer model, where each node periodically exchanges monitoring information with a dynamically selected subset of peers.

5.2.1 Message Propagation Process

The Information Dissemination Controller initiates and manages gossip-based communication. Each node maintains a local state repository, which stores recently collected monitoring data. At predefined intervals, known as the gossip rate, a node selects a random subset of peers and shares its latest state update. The number of peers contacted in each gossip round is determined by the gossip count parameter, which introduces a controlled level of redundancy while avoiding excessive network overhead. A more detailed sequence is depicted in Figure 5.2.
When a node initiates a gossip round:

1. The node extracts a snapshot of its local monitoring state from the SR, including metrics such as CPU usage, memory consumption, and network activity.
2. It randomly selects gossip count peers from its known list of active nodes. The random selection ensures that information spreads non-deterministically, preventing predictable network congestion and increasing resilience to failures.
3. The node sends a gossip message containing its monitoring update to the selected peers. Each message includes metadata such as timestamps and node identifiers, allowing recipients to determine whether the received data is newer, identical, or outdated compared to their state.
4. Upon receiving a gossip message, a peer compares the received monitoring data with its local state. If the received data is more recent or contains new information, the peer updates its local state repository accordingly.
5. The recipient then propagates the updated information to a new subset of peers in the next gossip round, following the same probabilistic selection process.

This process continues iteratively, allowing information to spread exponentially across the network. The design ensures that even if some messages are lost due to network failures

or node departures, the redundancy introduced by multiple gossip rounds guarantees that all nodes eventually receive the latest monitoring data.

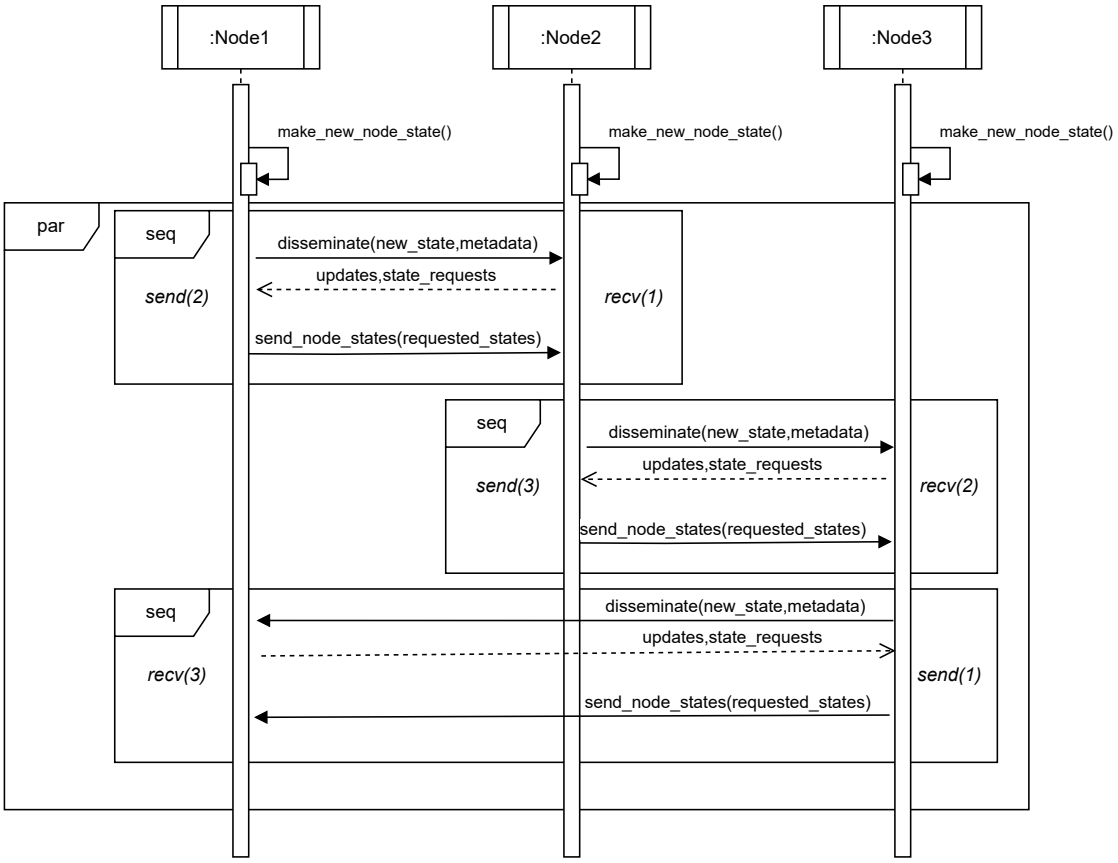


Figure 5.2: Sequence diagram of 3 Nodes gossiping [IFTB24].

5.2.2 Randomness in Gossip Propagation

The gossip count parameter determines how many peers each node contacts per round. Instead of using a fixed, deterministic set of clients, DEMon employs a randomized selection process, ensuring that the gossip topology remains dynamic and evenly distributed. This randomness prevents:

- Network bottlenecks, as no single node consistently receives disproportionate updates.
- Partitioning effects, where isolated clusters form due to static peer selection.
- Single points of failure, as message loss at one node does not prevent the information from reaching others through alternative paths.

The higher the gossip count, the faster the information spreads, but at the cost of increased message complexity. Conversely, a lower GC reduces network overhead but may slow down convergence, requiring more gossip rounds for all nodes. The optimal gossip count is experimentally determined, balancing convergence efficiency and network resource utilization.

5.2.3 Information Aggregation & Convergence

DEMon employs a distributed aggregation strategy to maintain accurate monitoring data across all nodes. Since each node periodically receives updates from multiple peers, it must determine which data to retain, discard, or propagate further. This is achieved through:

1. Timestamp-Based Prioritization: Newer updates always replace older ones to ensure that stale data does not persist in the system.
2. Selective propagation: Nodes only forward updates if they contain new or previously unseen information, reducing redundant message exchanges.
3. Multi-round convergence: Due to the probabilistic nature of gossip, the information does not spread instantaneously. Instead, it gradually converges over multiple gossip rounds, reaching all nodes with high probability.

The efficiency of this epidemic-style dissemination makes DEMon highly adaptive to dynamic network conditions. Even when nodes frequently join or leave, the gossip protocol ensures that monitoring data remains up-to-date and widely available, maintaining a robust and self-healing monitoring infrastructure.

5.3 Key Design Considerations

The need for scalability, efficiency, and trust in decentralized edge monitoring shapes the design of DEMon. Ensuring that data dissemination, retrieval, and resource usage remain optimal requires balancing several architectural and algorithmic trade-offs.

A primary consideration is message complexity within the gossip protocol. While increasing the gossip count accelerates convergence, it also increases network overhead. The system optimizes this by dynamically adjusting peer selection randomness to prevent unnecessary duplication while maintaining fast data propagation. This ensures that information spreads efficiently without overloading the network.

Another critical aspect is data retrieval accuracy. Since monitoring information is stored in a fully decentralized manner, nodes must verify the consistency of retrieved data without relying on a leader. The Leaderless Quorum Consensus protocol determines the minimum number of matching responses required for a monitoring query to be considered

reliable. This prevents inconsistencies while ensuring that monitoring results remain timely and verifiable.

Compared to fog-based solutions, which rely on aggregation nodes to process monitoring data centrally, DEMon's peer-to-peer approach eliminates dependency on intermediate layers. However, this design also introduces the challenge of reducing redundant messaging while ensuring that each node eventually receives accurate updates. The system mitigates this by limiting unnecessary retransmissions through intelligent state comparison and adaptive dissemination strategies.

Another key consideration is computational efficiency, as edge devices often have limited processing capabilities. DEMon minimizes its resource footprint by optimizing memory usage in the State Repository and avoiding excessive data exchanges. The system's containerized architecture further ensures lightweight deployment and compatibility with heterogeneous edge hardware, allowing it to operate in diverse environments without modifications.

By addressing these challenges, DEMon balances scalability, reliability, and minimal resource consumption ensuring that it remains functional in large-scale, dynamic edge deployments.

5.4 Implementation Details

DEMon [1] is implemented as a lightweight, containerized monitoring system designed for decentralized edge environments. It is built using Python, with Flask for API communication, psutil for system monitoring, and SQLite for persistent local storage. The modular architecture ensures scalability, efficient data dissemination, and reliable monitoring retrieval.

5.4.1 DEMon APIs

Each node runs a DEMon Agent (DA) that provides monitoring, dissemination, and retrieval services. The agent exposes RESTful APIs using Flask, enabling nodes to communicate with each other and allowing external applications to query monitoring data. The key Endpoints responsible for controlling a node are:

- `/start_node`: To initiate a new node or component in the monitoring system, a `POST` request needs to be sent to the `/start_node` endpoint. The payload for this request should be in JSON format and include the necessary configuration parameters. A example payload can be seen as a JSON here 1.
- `/stop_node`: Instantly stop the whole gossiping and all other actions from the DEMon client.

[1] `https://github.com/hpc-tuwien/DEMon`, accessed 07-02-25

```
{
  "node_list": [],
  "gossip_count": 3,
  "gossip_rate": 1,
  "database_address": "db_address",
  "monitoring_address": "monitoring_address",
  "client_port": "4000",
  "node_ip": "ip_of_host",
  "is_send_data_back": 0,
  "push_mode": 0
}
```

Listing 1: Example payload to start DEMon on a node per API call.

- `/reset_node`: Resets the node and all its hyperparameter.
- `/hello_world`: Once called the server answers if its healthy.
- `/get_recent_data_from_node`: Sends the most recent data stored in the `SR`.
- `/get_recent_data_from_node`: Send back the whole `SR`, which contains all the monitoring data the node has stored.
- `/metadata`: Returns the whole metadata of a node.

5.4.2 Information Dissemination

The exchange of messages between the nodes can be roughly divided into 2 main tasks. One of these is the sending of self-stored data, the other is the receiving.

Sending

Sending data cannot be regarded exclusively as sending. In general, it is understood here that current data is sent from node a (the sender) to node b (the receiver). In the process, however, messages are also received by the receiver (node b). The pseudo code 5.1 gives a more detailed insight into the process.
Every gossip round (at an interval of gossip_rate) the current monitoring data of the node is saved in the State Repository. Recipient nodes are then randomly selected for each gossip count to which the new data is to be sent. The first step is to check whether the node is still online (whether the failure_treshold has been exceeded); if this is not the case, the sender first sends its own metadata from the SR to the recipient. In response, the sender receives all current data that has been compared with its own SR and also a list of data that is to be sent. As the sender has previously updated its own data, at least one (its own) update is sent to the receiver. This now continues for all selected

receivers nodes. In a theoretically closed system that runs synchronously, the sender is up to date with its receivers.

Algorithm 5.1: Information Dissemination Controller - Send Node States [IFTB24]

```
1  for every gossip_rate do
2      s ← make_new_node_state(); ▷ Collect monitoring data and compute digest
3      SR ← store_node_state(s);
4      nodes ← select_nodes_to_gossip();
5      for n ∈ nodes do
6          if |n.U| ≥ failures_threshold then
7              SR ← delete_node(n.id);
8          else
9              SR_metadata ← get_SR_metadata();        ▷ Node IDs and counters
10             response ← disseminate(n.id, s, SR_metadata);
11             if response then
12                 updates, requests ← parse_response(response);
13                 for u ∈ updates do
14                     SR ← store_node_state(u);
15                 end
16                 requested_node_states ← get_requested_node_states(requests);
17                 send_node_states(n.id, requested_node_states);
18             else
19                 SR ← update_unreachable_node(n.U);
20             end
21         end
22     end
23 end
```

Receiving

The receiver side is shown in the algorithm 5.2. If the node is randomly selected as a receiver in the gossip algorithm, it first receives the metadata of the sender in order to compare it with its own state repository. A distinction is made in 2 cases: The metadata for a specific node is more up-to-date in the recipient node or not. If they are not up to date, the node id from which the data is out of date is added to the requests. If they are more current, the data package is added to the updates. Finally, both packets are sent back to the sender and both sender and receiver have the same up-to-date data from all known nodes in the network.

Algorithm 5.2: Information Dissemination Controller - Receive Node States [IFTB24]

```
1  while true do
2  |  sender_node_state, SR_metadata ← parse_received_message();
3  |  SR ← store_node_state(sender_node_state);
4  |  for node ∈ SR_metadata do
5  |  |  current_node_counter ← get_node_counter(node.id);
6  |  |  if node.counter > current_node_counter then
7  |  |  |  requests ← node.id;
8  |  |  else
9  |  |  |  updates ← get_node_state(node.id)
10 |  |  end
11 |  end
12 |  send_updates_and_requests(updates, requests);
13 end
```

5.4.3 State Repository

The state repository forms the core of the framework presented. It is an in-memory database that ensures fast access. An entry for a node with the respective timestamp as a key is shown in this JSON 2. The respective subcategories store different information about the system:

- `nodeState` contains all information to reach the node via the network, such as the internally assigned id, the global ip and also the port via which the API can be reached.
- `hbState` contains its own timestamp to record the time of each measurement. The failureCount and failureList entries are used for health monitoring, as is "nodeALive".
- `appState` contains all relevant monitoring data, such as cpu, memory, network and storage information in this case.

5.4.4 Data Retrieval as a Client

The principle of Leaderless Quorum Consensus, which is mapped in the pseudo-code 5.3, was applied to ensure a trustworthy and efficient data query from outside. As each node should have data from every other node in its memory after convergence, data can of course also be queried from a single node. However, as this data can be compressed but also outdated, the LQC offers an efficient alternative. The metadata is queried from a certain number (quorum size) of randomly selected nodes. The timestemp and digests are compared for the desired information. If all the information about the metadata from

```
{
 "counter": 3,
 "round": 3,
 "nodeState": {
     "id": 1,
     "ip": 138.217.156.40,
     "port": 4000
 },
 "hbState": {
     "timestamp": 0,
     "failureCount": 0,
     "failureList":[],
     "nodeAlive": 1
 },
 "appSate": {
     "cpu": 35,
     "memory": 50,
     "network": 3400,
     "storage": 15000
 },
 }
```

Listing 2: Example of a State Repository entry

the nodes matches, a consensus is formed that the data is relatively up-to-date and also trustworthy.

5.4.5 Monitoring Metric Collection

There is countless data that can be monitored in heterogeneous edge environments. This work is limited to collecting CPU, RAM, network and storage data. It should be noted that these metrics are representative and, depending on the technology, all metrics can be easily integrated into the DEMon framework. The metrics listed were recorded as follows:

- `CPU`: The `psutil.cpu_percent()` method measures the percentage of CPU usage over a specified interval. This allows for real-time monitoring of CPU performance, helping to identify resource-intensive processes and avoid performance degradation.
- `RAM`: `psutil.virtual_memory().percent` tracks the percentage of system memory in use. By monitoring this metric, DEMon can assess the memory load

Algorithm 5.3: Leaderless Quorum Consensus Protocol [IFTB24]

```
1  query_nodes ← select_random_query_nodes(quorum);
2  R ← query_metadata(query_nodes);                    ▷ Node IDs and counters
3                                                      ▷ Provides a notion of weak consistency
4  if compare(R.timestamp) is true then
5  |                                                   ▷ Ensures data trustworthiness
6  |  if compare(R.digest) is true then
7  |  |  return query_data();
8  |  else
9  |  |  go to 1;
10 |  end
11 else
12 |  go to 1;
13 end
```

on the system, which is crucial for preventing memory bottlenecks and ensuring smooth operation.

- `Network`: The sum of `psutil.net_io_counters().bytes_recv` and `psutil.net_io_counters().bytes_sent` provides the total data received and sent over the network. This helps monitor network traffic, identify abnormal data usage, and optimize network-related performance.

- `Storage`: `psutil.disk_usage('/').free` measures the available storage space on the root directory. Tracking this value is essential for ensuring that there is sufficient disk space for applications and preventing errors due to a full disk.

5.4.6 Transition from Kubernetes to Docker

Challenges with Kubernetes

In the UCC 2022 [IFDB22] implementation, DEMon was deployed using Kubernetes [2], where each pod represented an individual monitoring node. This approach allowed for containerized execution, dynamic orchestration, and automatic scaling, leveraging Kubernetes' built-in resource management and networking mechanisms. However, despite these advantages, several challenges emerged when deploying a fully decentralized monitoring system within a Kubernetes-based testbed.

- A key limitation of Kubernetes in this context was its scalability on a single server. A single-node Kubernetes cluster could not handle more than X pods, as each pod introduced additional CPU, network, and storage overhead, limiting the number of active monitoring nodes that could be deployed. Unlike a lightweight containerized

[2] `https://kubernetes.io/`, accessed 07-02-25

system, Kubernetes enforces strict resource isolation per pod, increasing overhead and making large-scale deployment impractical on a single machine. Multiple physical servers would have been required to scale further, effectively introducing a hierarchical infrastructure where Kubernetes assigns workloads across nodes. This directly contradicted DEMon's fully decentralized architecture, as it would have created an implicit aggregation layer where Kubernetes centrally manages nodes. Since DEMon is designed to operate without hierarchy, introducing a Kubernetes-based multi-server setup could have influenced test results by disrupting the natural peer-to-peer behaviour expected in a decentralized network.

- Another major issue was restricted access to individual monitoring nodes, which interfered with the information retrieval mechanism in DEMon. Kubernetes does not allow direct external access to pods, requiring all communication to pass through its central API system, meaning queries could not be sent directly to individual DEMon nodes. Instead, all monitoring queries had to be handled via Kubernetes' internal service discovery and routing mechanisms, introducing additional layers of abstraction. This conflicted with DEMon's decentralized design, where nodes should be able to communicate directly without relying on a central registry. Additionally, the Kubernetes networking model required explicit service definitions and cluster IP assignments, further complicating configuration and making inter-container messaging inefficient. These constraints made it difficult to implement and evaluate DEMon's Leaderless Quorum Consensus protocol, which relies on direct peer-to-peer queries for monitoring data retrieval.
- The overall complexity of managing Kubernetes-based deployments also proved challenging. Kubernetes enforces strict pod lifecycle management, persistent volume handling, and inter-service communication policies, requiring additional setup to maintain consistency across multiple nodes. Since DEMon requires each node to store, share, and retrieve monitoring data autonomously, adapting its architecture to Kubernetes' centralized state management added unnecessary overhead. The orchestration layer imposed by Kubernetes interfered with DEMon's need for direct, lightweight deployments, making it difficult to maintain the flexibility and adaptability required in a fully decentralized edge network.

Migration to Docker-Based Deployment

To address these limitations, the recent implementation of [IFTB24] changed to a pure Docker [3] based deployment, where each monitoring node runs as an independent Docker container without Kubernetes orchestration. This change significantly improved scalability, efficiency and direct node accessibility, ensuring that DEMon's testbed remained fully decentralized and free from external scheduling constraints.

One key advantage of switching to Docker was improved scalability on a single machine. Unlike Kubernetes, Docker allows launching hundreds of containers on a single physical

[3] `https://www.docker.com/`, accessed 07-02-25

server without the excessive resource overhead introduced by Kubernetes' pod management system. Since Docker containers share more underlying system resources, they are significantly lighter regarding CPU, memory, and network consumption, enabling larger-scale deployments without additional infrastructure. This ensured that more DEMon nodes could be deployed and tested in a fully decentralized environment while maintaining realistic edge computing constraints.

The transition also eliminated hierarchy, keeping DEMon's testbed fully peer-to-peer without requiring a centralized cluster manager. Since Docker does not impose any built-in orchestration beyond simple container execution, each DEMon instance remains independent, allowing monitoring nodes to function autonomously as initially intended. Unlike Kubernetes, which schedules workloads based on cluster-wide resource availability, Docker containers run without external scheduling influence, ensuring that all nodes participate equally in monitoring and data dissemination. This made it possible to test true decentralization without worrying about Kubernetes' hierarchical decision-making affecting information flow.

Finally, the Docker-based deployment enabled direct node communication and simplified configuration, ensuring that each DEMon instance could be queried independently. Unlike Kubernetes, which requires all inter-container communication to pass through an internal API system, Docker containers can interact directly via user-defined networks, preserving DEMon's original communication model. This made it possible to accurately evaluate DEMon's LQC protocol, as monitoring queries could now be resolved directly between nodes without intermediary services. Removing Kubernetes' service discovery and pod abstraction layers further streamlined the deployment, reducing unnecessary complexity and simplifying testbed management.

5.4.7 Specific Requirements

Node Requirements

The following items provide the detailed python requirements to run a DEMon Client.

- `Flask==2.0.3`: In order to run a lightweight API-Server on each Node.
- `psutil==5.9.0`: Handles the Hardware monitoring.
- `requests==2.27.1`

Emulation Requirements

In order to run the full emulation experiments, the following python libraries 5.1 are necessary.

Name	Purpose
`concurrent.futures`	Library for asynchronous execution using futures.
`configparser`	Parser for configuration files in INI format.
`json`	JSON encoder and decoder for Python.
`random`	Library for generating random numbers.
`sqlite3`	Library for SQLite database management.
`time`	Time-related functions and utilities.
`docker`	SDK for managing Docker containers in Python.
`socket`	Low-level networking interface.
`requests`	Library for sending HTTP requests.
`traceback`	Library for handling and printing exceptions.
`queue`	Queue data structure for multithreading.
`threading`	Library for working with threads.
`Flask`	Micro web framework for Python.
`joblib`	Library for parallel computing and caching.
`logging`	Standard library for logging messages in Python.

Table 5.1: Python Packages for Emulation Experiments

5.5 Overall System Workflow

This section explains how a general emulation via docker experiment works. Figure 5.3 shows the rough system structure and how the systems interact with each other. First, the executor of the operation enters the desired system and hyper parameters in a config file. As soon as docker is started on the executing server, a request can be sent to the emulation server, which starts the setup for the system. First, the desired number of nodes is created as a Docker container and built with the current DEMon image. These are continuously checked so that the experiment can be started as soon as they are online and the respective DEMon instances are running. This is done with a series of parallel requests to the individual DEMon agents. Once these are started, they start exchanging messages with each other: gossiping. Each of these messages is also forwarded to the emulation system to enable further data analysis, as in this work. The entire system status is updated with each message. Depending on the configuration, the expereiemtn is canceled as soon as the system has converged. Parallel requests are sent to reset the individual nodes and thus also their data. Individual nodes that do not respond to the reset are switched off and replaced by new instances (in this case containers). Now the experiment starts again with different parameters if necessary. If the system size changes, new instances are created and the list of nodes is updated.

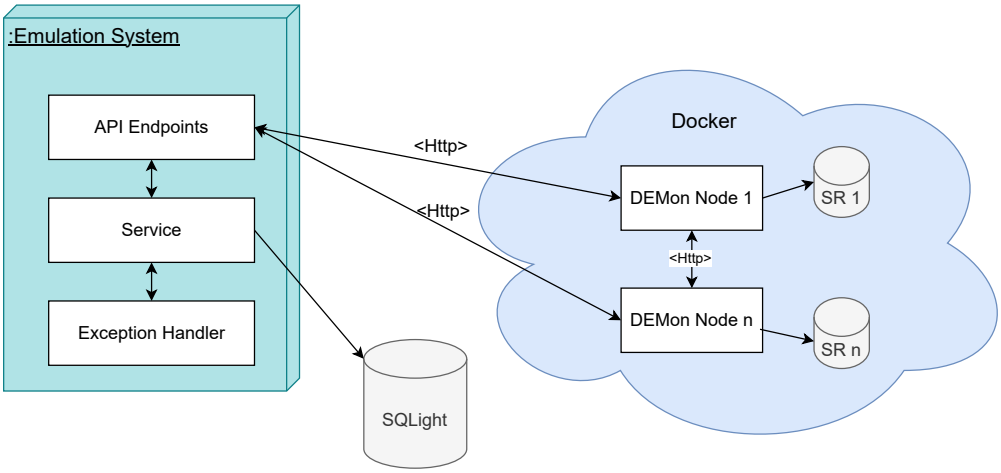


Figure 5.3: Emulation System Overview

CHAPTER 6

Evaluation

This chapter presents this thesis's empirical evaluation, assessing the framework's performance, efficiency, and scalability in a decentralized edge environment. The review is structured to align with the defined research objectives, focusing on key benchmarking parameters such as convergence speed, network overhead, and resource consumption. A detailed analysis of the system examines the internal behaviour of DEMon, while a comparative analysis highlights its advantages and trade-offs against existing decentralized monitoring solutions. The findings are then discussed in relation to scalability, fault tolerance, and real-world applicability, followed by a reflection on the limitations of the current approach and potential areas for improvement.

6.1 Objectives

One of the aims of the thesis is to answer the mentioned research questions (section 4.1) in detail. RQ1 is discussed in the sections 6.3.1 and 6.3.3. The answer to RQ2 can be found in section 6.3.4. RQ3 is subsequently considered under 6.3.1. The answers to RQ4 are analyzed in detail in section 6.3.5.

6.2 Benchmarking- and Hyperparameter

This section defines the key performance metrics used to control, but also to evaluate DEMon.

6.2.1 Hyperparameter

The evaluation framework allows for the configuration of key hyperparameters that define each experiment. These parameters are set in a config.ini file, ensuring repeatability and

controlled variations across different runs. The primary hyperparameters influencing the evaluation are

- `gossip_count`: Specifies how many target nodes a node selects for each gossip round
- `gossip_rate`: Defines the interval between consecutive gossip rounds, affecting how frequently nodes exchange monitoring data.
- `system_size`: Determines the total number of nodes in the experiment, providing insights into scalability and system behavior at different network sizes.

Other adjustable settings, while not directly impacting the evaluation, allow for additional control over the experiment execution. These include whether the system should continue running after convergence, how many runs should be performed (fixed to three for this thesis), whether monitoring data should be pushed to an SQL database, and whether the query logic should be activated to simulate an external client retrieving monitoring data. These configurations help maintain a structured experimental setup.

6.2.2 Benchmarking Parameter

The evaluation of DEMon is based on several quantitative performance metrics, each providing insights into different aspects of the system's behavior and efficiency. These benchmarking parameters are measured throughout the experiments to assess scalability, convergence speed, network overhead, and storage efficiency.

- **Number of Messages:** Tracks the *total number of messages* exchanged until a predefined system state is reached, indicating the communication overhead required for information propagation.
- **Number of Rounds:** Measures how many *gossip rounds* are necessary to achieve a specific system state, reflecting the protocol's efficiency in iterative message dissemination.
- **Time:** Records the *total time required* to reach a certain experimental milestone, such as full system convergence.
- **Storage:** Captures the *total storage consumption* across all nodes and the *storage footprint per node* at specific time intervals, evaluating the impact of decentralized data retention.
- **Bandwidth:** Analyzes the *average bandwidth consumption per node*, providing insights into the network load imposed by the monitoring system.
- **Age of Information:** Determines the *average age of retrieved monitoring data*, measuring how fresh the information remains over time.

- **New Data:** Quantifies how many *new data points* each node receives that lead to an new entry in the *State Repository*, helping to evaluate the rate of meaningful data propagation.
- **Fresh Data:** Quantifies how many *fresh data points* each node receives that lead to an update in the *State Repository*.
- **Convergence:** Defines the *predefined system state* where each node has successfully received and stored at least *one monitoring update from every other node*, marking complete network synchronization.

6.3 System Analysis

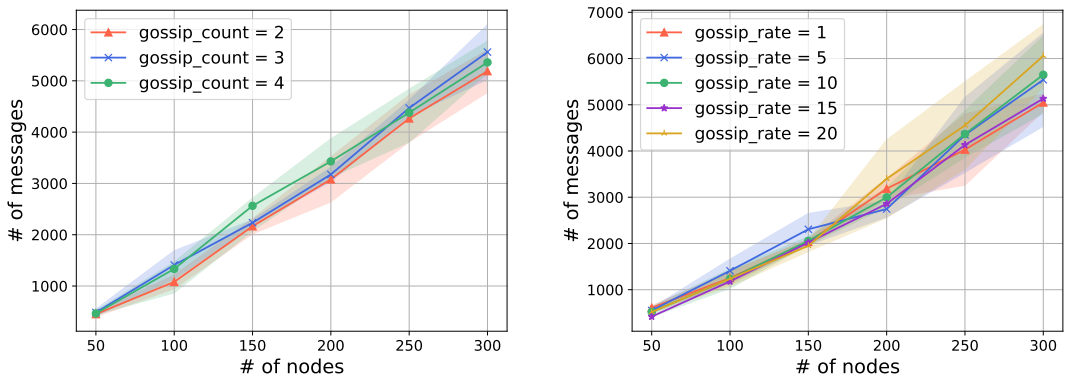
This section provides a detailed evaluation of DEMon's performance using key benchmarking metrics. It offers insights into the system's operation under various configurations and network conditions. The analysis is organized into four main areas: message complexity 6.3.1, gossip rounds 6.3.2, time to convergence 6.3.3, and resource consumption 6.3.4. Each subsection addresses specific aspects of the system, emphasizing trends, trade-offs, and the influence of hyperparameters on overall performance. This section presents a detailed evaluation of DEMon's performance based on key benchmarking metrics, providing insights into how the system behaves under different configurations and network conditions. The analysis is based on metrics around messages needed for convergence, gossip rounds, time to convergence and resource consumption and offers a comprehensive understanding of the system's capabilities.

6.3.1 Message-Based Analysis

Since each node interacts with its peers via API calls, every exchanged message is collected by the testing framework. The total message count provides an abstract measure of the resources required for the system to reach convergence. Convergence, as previously defined, represents a key system state that serves as a baseline for analyzing the system's efficiency. In this thesis, convergence is achieved when each node in the network has received monitoring data from all other nodes. Information dissemination does not require direct communication between every node pair. For example, if Node 1 receives a message from Node 2 and subsequently sends a new message to Node 3, then Node 3 will have data from Node 1, Node 2, and itself. This means that nodes can indirectly propagate information, ensuring that the system converges without requiring all nodes to communicate directly with each other.

Figure 6.1 provides key insights into how the `gossip_count` and `gossip_rate` parameters influence the total number of messages exchanged until convergence in the docker-based testbed. The results align with expectations: neither hyperparameter significantly alters the total number of messages required for convergence. In the case of Figure 6.1b (`gossip_rate`), the explanation is straightforward. The total number of messages sent is independent of the rate at which those messages are transmitted. Since DEMon

operates based on a randomized dissemination model, altering the `gossip_rate` merely affects the timing of exchanges, not the overall message complexity. The standard deviation, represented by the lighter shaded area, exhibits some variability due to the inherent randomness of gossip-based communication. However, with an increasing number of runs ($n \to \infty$), the total message count remains stable, converging toward a consistent linear trend across different gossip_rate values. As expected, the only significant factor affecting the message count is system size, which in this experiment is scaled from 50 to 300 nodes in increments of 50. In contrast, Figure 6.1a (`gossip_count`) presents a slightly different scenario. In theory, increasing `gossip_count` should result in a higher total message count, as more messages are sent per gossip round. If gossip_count were sufficiently large, the protocol would behave more like flooding, leading to redundant message exchanges and reducing the efficiency of information dissemination. However, Figure 6.1a shows that for small values of `gossip_count` (`2-4`), the impact on the total message count remains negligible. This suggests that in the tested range, gossip-based dissemination remains efficient, and the system does not experience unnecessary message overhead due to minor variations in `gossip_count`.

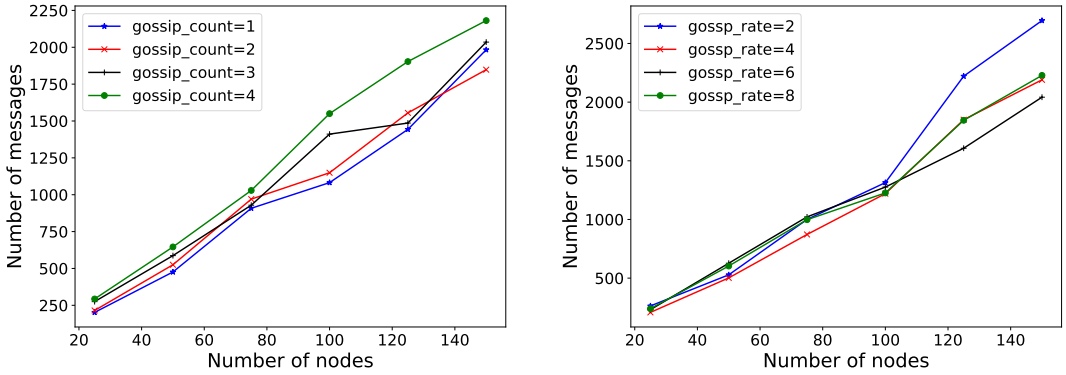


(a) `gossip_count` (b) `gossip_rate`

Figure 6.1: Influence of `gossip_rate` and `gossip_count` on the number of messages up to convergence in a docker-based framework [IFTB24]

A similar phenomenon can be observed in Figure 6.2. The overall trend remains the same, but this experiment was conducted on the previous version of the testbed, which was based on Kubernetes. As a result, the system size is limited to a maximum of 150 pods. Additionally, the case where `gossip_count` = 1 is included in Figure 6.2a, while Figure 6.2b presents a finer granularity for comparing different gossip_rate values. Once again, the results confirm that neither gossip_count nor gossip_rate has a significant influence on the total number of messages required for the system to converge. Instead, the total message count remains primarily correlated with the system size. This further reinforces the findings from the Docker-based experiments, demonstrating that the underlying system architecture does not fundamentally alter the relationship between system size

and message complexity.



(a) `gossip_count` (b) `gossip_rate`

Figure 6.2: Influence of `gossip_rate` and `gossip_count` on the number of messages up to convergence in a kubernetes-based framework [IFDB22]

Leaderless Quorum Concensus

To evaluate query latency, the number of messages required to retrieve monitoring data for any given node was measured. In this experiment, 100 queries were conducted under different failure rates, where each query requested the utilization metrics of a randomly selected node. The querying mechanism followed the process outlined in Algorithm 3, with the quorum size set to 3. For each query, three randomly chosen nodes were contacted in parallel. When a node received a request, it accessed the requested node ID (IP address) in its local storage and returned the corresponding monitoring data in the predefined format. The failure rate was varied between 0% and 90% in 10% increments, with each configuration randomly disconnecting the corresponding percentage of nodes. Despite these failures, the requested information was consistently retrieved without query failures. Since each node maintains a copy of all other nodes' monitoring data, even with 90% node failures, the requested information remained accessible. The number of messages required per query ranged from a minimum of 3 (ideal case, equal to quorum size) to a maximum of 148, with an average of 10.45 and a median of 3 [IFTB24].

6.3.2 Round-based Analysis

The round-based metric provides new insights into the system. As previously mentioned, a round refers to a cyclic event in which nodes exchange messages within the system. In the experiments, each message is logged along with the round in which the sending node is currently operating. This allows for a detailed analysis of message dissemination across different rounds. It is important to note that each node acts independently, meaning that a round does not occur simultaneously for all nodes in the system. Due to this

decentralized nature, there is no global synchronization of rounds, and nodes may progress through rounds at different times. Any delays caused by longer message transmission times or other factors are not explicitly considered in the following analyses.

As discussed earlier, the hyperparameters `gossip_count` and `gossip_rate` have little to no influence on the total number of messages sent. However, their impact on the number of rounds required for convergence is different. While `gossip_rate` also affects the number of rounds until convergence, `gossip_count` plays a significant role in this aspect.
As clearly shown in Figure 6.3, there is a negative correlation between the number of rounds to convergence and the `gossip_count` parameter: the higher the `gossip_count`, the fewer rounds are needed for convergence. This system property is also intuitive, as `gossip_count` defines how many messages a node sends per round. Assuming that the total number of messages until convergence remains approximately constant, a clear linear dependency emerges. This dependency becomes even more pronounced as system size increases. For example, at a system size of 300, doubling the `gossip_count` nearly halves the number of rounds required for convergence. This demonstrates that adjusting the `gossip_count` can directly influence the speed of information dissemination in the system.

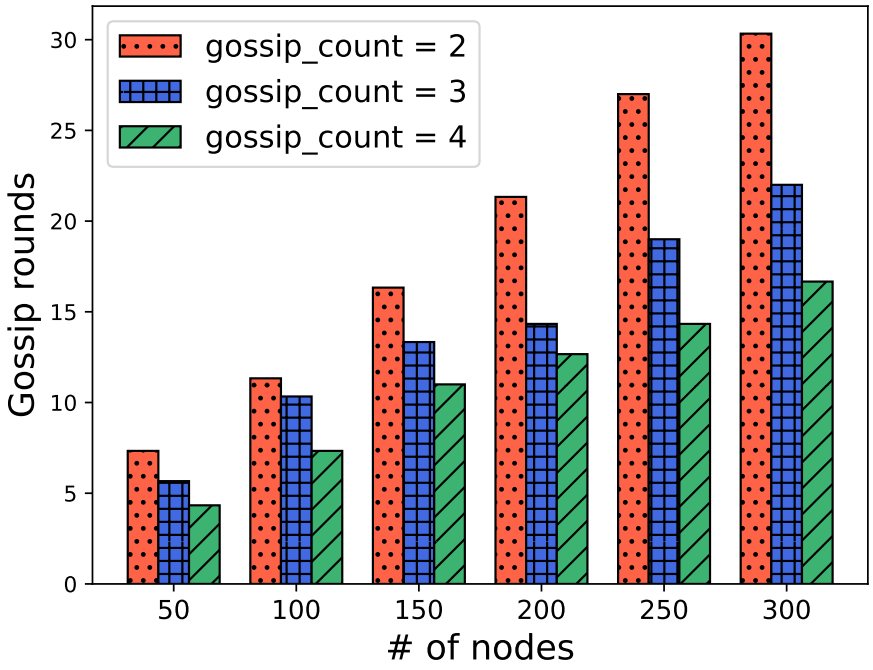
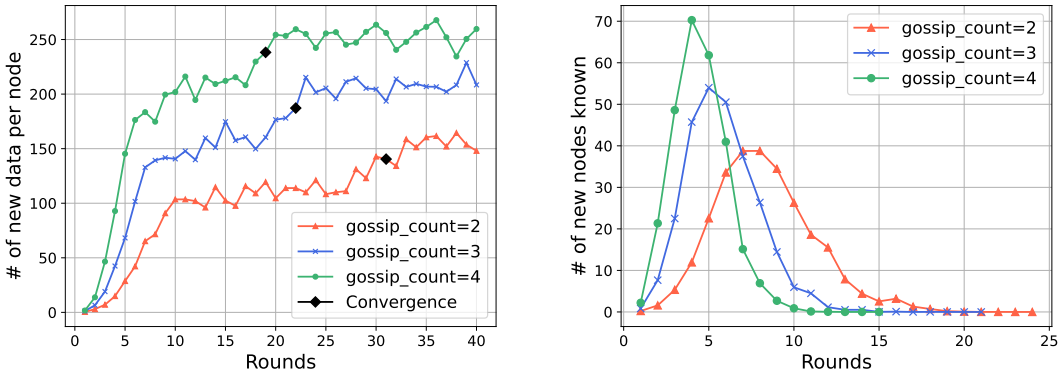


Figure 6.3: Influence of gossip_count on the convergence round [IFTB24].

Information Dissemination

Since rounds define a system-wide state that evolves individually for each node and progresses asynchronously over time, they provide new insights into the spread of information, as illustrated in Figure 6.4. As previously mentioned, every message sent by a node is logged, along with the round in which it was transmitted. This enables a deeper analysis of how data propagates throughout the system. Two key perspectives emerge from this data exchange: `Fresh_Data` and `New_Data`. Fresh_Data represents the total number of data points in a given round that update the local State Repository of a node—meaning all information that was modified through interactions with other nodes. Within `Fresh_Data`, a subset of data points is classified as `New_Data`. These refer specifically to data received from a node that was previously unknown to the receiving node. By distinguishing between `Fresh_Data` and `New_Data`, it becomes possible to quantify the rate at which nodes receive new information versus how frequently existing data points are updated through repeated exchanges. This provides valuable insights into the dynamics of information dissemination and the efficiency of the gossip-based protocol in achieving system-wide convergence.

Figure 6.4a shows the average `Fresh_Data` per round per Node, from the start (Round = 0) up to the 40th round, and the influence of gossip_count on the number of updated data points. It is clearly visible that at the beginning, a large number of data points in the SR are updated, as nodes initially only store their own data locally. Depending on the `gossip_count`, the system converges after a certain number of rounds. This further highlights why convergence is such a crucial state for the system. From this point onward, the exchange of `Fresh_Data` follows a stable and nearly linear pattern. This behavior occurs because, at convergence, each node that sends a message has already stored at least one data packet from every other node in its SR. If this data packet is more recent than the version held by another peer, it will be updated as new `Fresh_Data` when the message is transmitted.
Figure 6.4b illustrates the distribution of `New_Data`, which refers to data from nodes that were previously unknown to the receiving node at the time of message reception. The observed trend clearly indicates how many new nodes are, on average, discovered per node during the experiment. The peak and the steepness of the initial curve are strongly influenced by the gossip_count. A clear correlation emerges: the higher the `gossip_count`, the more new data (as well as fresh data) is exchanged per round. The subsequent decline in the curve can be explained by the fact that, after a certain number of rounds, most nodes have already been discovered by others, reducing the number of newly received datasets from previously unknown nodes. Additionally, it becomes evident that in the final rounds leading up to convergence (approximately the last five rounds), almost no `New_Data` is found. This is a characteristic property of randomized selection: the process of discovering new nodes naturally diminishes as all nodes become known to each other. Once this state is reached, the system is considered converged.



(a) FD (b) ND

Figure 6.4: Fresh_Data and New_Data per round [IFTB24]

6.3.3 Time-based Analysis

In addition to system-specific parameters, the time-based analysis represents a framework-independent metric. Since the system's performance can be significantly influenced by both software and hardware characteristics, the absolute time values in seconds are of secondary importance in the following analysis. Instead, the focus is placed on examining the impact of hyperparameters on time-related performance. Additionally, this part provides a detailed analysis of the Age of Information, evaluating how retrieved monitoring data remains up-to-date over time.

As mentioned in Section 6.3.1, the parameters `gossip_count` and `gossip_rate` do not influence the total number of messages required for convergence. However, as shown in Section 6.3.2, these system settings have a direct impact on the number of rounds until convergence. Since a round represents a cyclic event in which data is transmitted to `gossip_count` other nodes at intervals defined by `gossip_rate`, it is intuitive that both parameters also influence the absolute convergence time. Figure 6.5 clearly illustrates the impact of these parameters. The x-axis represents different `gossip_rate` configurations, while the various colored plots correspond to different `gossip_count` values. The system size is fixed at 300 nodes. Several trends emerge: as `gossip_rate` increases, the time required for the system to converge also increases, which aligns with the definition of `gossip_rate`. Additionally, the total convergence time decreases as gossip_count increases. In general terms, the shorter the interval and the more nodes receive data per interval, the shorter the convergence time. Interestingly, the observed influence is significantly lower than expected. In theory for example, for `gossip_rate = 1`, increasing `gossip_count` from 2 to 4 should halve the time required for convergence, as only half the number of rounds would be needed. However, the test results show a much smaller reduction. The reasons for this discrepancy are varied and beyond the scope of this thesis.

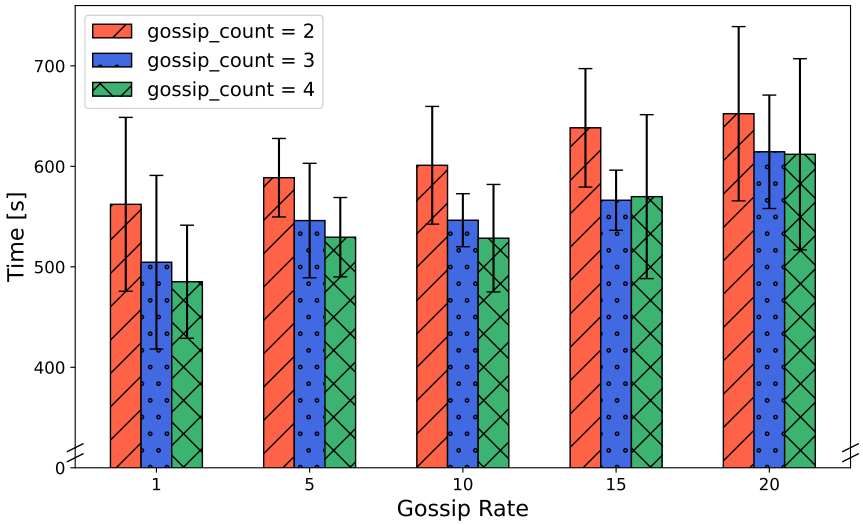


Figure 6.5: Influence of gossip_count and gossip_rate on time till convergence [IFTB24].

Age of Information

The Age of Information (AoI) is a statistically computed value that represents the freshness of data within the system. Each node appends a timestamp to its monitoring data, allowing both the measurement time and data recency to be determined. When these data points are forwarded to other nodes, it becomes possible to evaluate how up-to-date the information at any given node is. This mechanism is also used to determine when monitoring data should be updated. This provides a clear measure of how much time has passed since the last relevant data update, with higher values indicating increasingly outdated data. Figure 6.6 illustrates the global AoI over a given time span. As expected, a linearly increasing trend is observed, with only minor variations depending on gossip_count. The increasing slope occurs because the AoI is computed relative to the continuously advancing local time counter of each node. Since this counter is always increasing, the AoI naturally rises over time.

6.3.4 Resource Usage

The resource overhead of monitoring systems is a critical factor, as in most cases, not only the monitoring framework itself but also other applications rely on the same shared computational resources. The goal of an efficient monitoring system is to maintain a balance between accurate measurements with precise data storage while minimizing computational resource consumption. In decentralized edge environments, resource efficiency becomes even more crucial due to hardware constraints and the need for distributed processing. Excessive CPU usage, memory consumption, or network bandwidth could

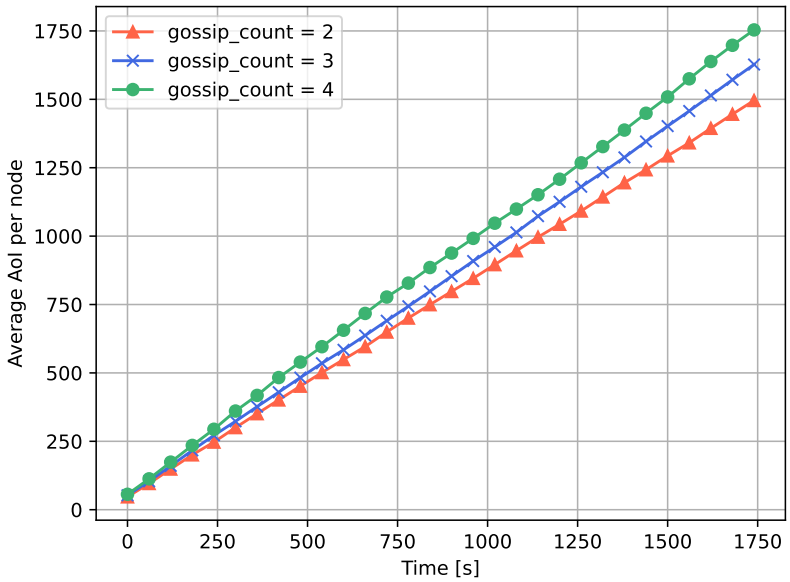


Figure 6.6: Influence of gossip_count on Age of Information.

negatively impact the performance of other running applications, leading to inefficiencies in system operation. To better understand the characteristics of DEMon, the following benchmarks provide an overview of the system's general resource requirements. These evaluations offer insights into the expected computational footprint, helping to assess the trade-offs between measurement accuracy and system efficiency. Additionally, the results illustrate how DEMon scales with increasing system size and whether it remains viable for real-world deployment in resource-limited environments.

Bandwidth

Figure 6.7 illustrates the impact of `gossip_count` on the bandwidth consumption per node. Only messages that contribute to the system's convergence were considered, which is why bandwidth usage decreases as the system approaches convergence. This scaling behavior is comparable to that observed in Figure 6.4: `New_Data`. When considering all exchanged messages, the trend follows a pattern similar to that in Figure 6.4a. A notable observation is that despite the number of messages doubling (comparing GC = 2 with GC = 4), the required bandwidth per node increases by only 40%. This finding highlights the scalability of DEMon, as higher `gossip_counts` are essential for maintaining efficient information dissemination in larger system sizes.

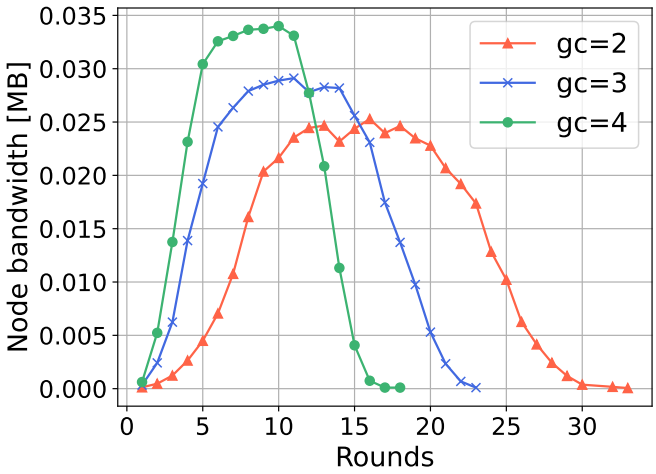


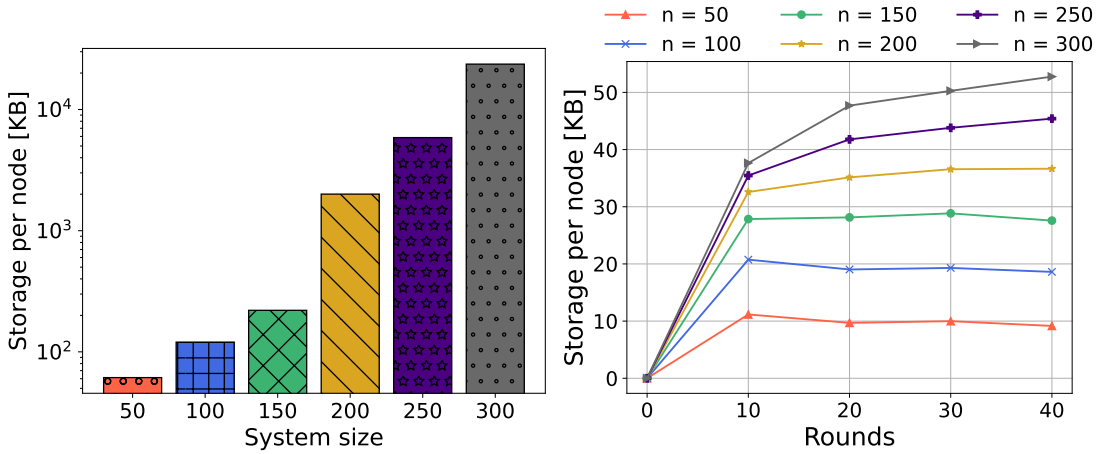
Figure 6.7: Bandwidth per node till convergence [IFTB24]

Storage

The framework supports two different modes for handling storage. If no dedicated storage is available, the entire State Repository, including all historical data, is stored in memory. Alternatively, if dedicated storage is provided in the form of an SQLite database, the system can periodically transfer the entire historical dataset every x rounds. In this case, the data is stored efficiently: redundant information is aggregated to optimize storage usage, and only changes in monitored resources are saved. Since every historical data point is retained and never deleted, Figure 6.8a illustrates the expected storage consumption per node depending on system size. It is important to note that each node stores the historical data of all other nodes, leading to exponential growth in storage requirements as the system size increases. Figure 6.8b further demonstrates the aggregation mechanism. In this configuration, all data is transferred every 10 rounds, and during this process, redundant entries are aggregated as described. The amount of data pushed to storage decreases after a certain point since redundant data is not stored repeatedly but instead referenced within the database, significantly improving storage efficiency.

Memory

Another important aspect is RAM consumption. Since DEMon is implemented as a prototype in Python, and each node runs its own HTTP server through the DEMon agent, slightly higher memory usage is observed compared to a more optimized implementation. Figure 6.9 illustrates how RAM consumption behaves when the entire State Repository is kept in memory. When the push mode is enabled, the memory usage remains stable at approximately 74 MB per node. It is also noteworthy that the choice of `gossip_count` does not have a significant impact on RAM usage. However, as DEMon continues to run and the SR grows over time, the variance in memory consumption increases. This effect



(a) Total Storage

(b) Pushed Storage

Figure 6.8: Total Storage after convergence and pushed storage every 10th round per system size [IFTB24]

is visible in the figure as the shaded region, representing the standard deviation, which expands with longer runtime.

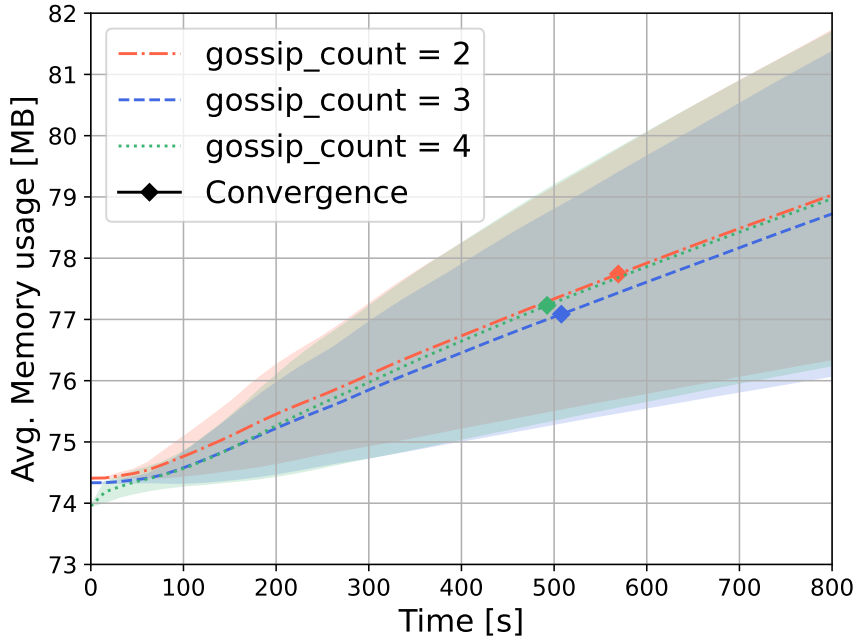


Figure 6.9: Memory Usage over time with SR completely in memory [IFTB24]

CPU

As the final metric, CPU utilization is analyzed. Figure 6.10 shows that CPU consumption remains consistently low, averaging around 0.65–0.7% and being independent of `gossip_count`. The initial fluctuations can be attributed to the use of a rolling average, which aggregates the last five measurement values. This further demonstrates the stability of the system and its low resource consumption, even in its prototype implementation.

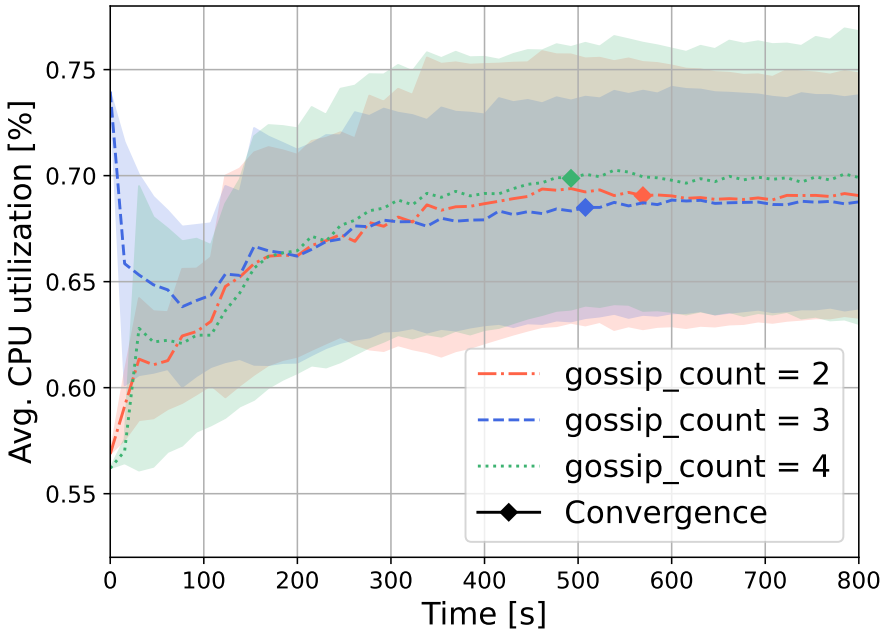


Figure 6.10: Rolling average of CPU-Utilization over time [IFTB24]

6.3.5 Comparative Analysis

To analyze DEMon not only within the defined hyperparameter space but also in comparison to a similar system, FogMon2 [1] was selected after thorough research. FogMon2 (detailed paper selection can be seen in the related work 3) was chosen because, among all publicly available monitoring tools, it adopts an approach most similar to that of DEMon. The key difference, however, lies in its hierarchical structure, as indicated by its name. Unlike DEMon's fully decentralized architecture, FogMon2 classifies nodes into Leader and Follower nodes. Each leader is assigned a group of follower nodes, which send their monitoring data exclusively to their designated leader. The leaders then communicate among themselves using a gossip-based approach with a fixed `gossip_count` of 2. A comparison of scalability approaches between DEMon and FogMon2 can be found in the next section 6.3.6 while Section 6.3.7 provides an evaluation of memory consumption.

[1] `https://github.com/di-unipi-socc/FogMon/tree/liscio-2.0`, accessed 07-02-25

A direct comparison of CPU usage was unfortunately not possible due to the technical characteristics of FogMon2.

6.3.6 Scalability Approach

As previously mentioned, FogMon2 classifies network nodes into Leader and Follower roles. The number of leaders in the system can be configured, with the default setting being $\sqrt{n}$, although configurations with $2 \cdot \sqrt{n}$ and $2/\sqrt{n}$ are also possible. Since the leaders communicate among themselves using a gossip-based protocol, the configuration with $2 \cdot \sqrt{n}$ most closely resembles the architecture of DEMon. In Figure 6.11, we compare different FogMon2 configurations with DEMon. For this comparison, the `gossip_count` in DEMon was set to 2, aligning with the gossip protocol used among leaders in FogMon2. Due to technical limitations, it was only possible to compare the systems up to a network size of 80 nodes. As a universal unit of comparison, we use the number of messages until convergence. For FogMon2, convergence is defined as the point when all leaders have received at least one monitoring update from every node in the system. This definition was chosen because, unlike DEMon, only selected nodes (leaders) are responsible for responding to monitoring queries. In contrast, DEMon reaches convergence when every node has received at least one data packet from every other node. In Figure 6.11, we clearly observe that for smaller system sizes (≤ 70 nodes), all configurations of FogMon2 reach convergence with significantly fewer messages, making it a more efficient monitoring solution in these cases. However, beyond 70 nodes, this trend changes, and only the configuration with fewer leaders maintains a lower message count. This is due to the leader gossip mechanism: with fewer leaders, the probability increases that they will quickly receive all necessary monitoring data, leading to faster convergence. However, reducing the number of leaders also increases centralization, reducing overall system robustness. Overall, DEMon exhibits better scalability than FogMon2 when the number of leaders increases. This result highlights the advantages of DEMon's fully decentralized approach, which maintains higher robustness and scalability compared to a hierarchical leader-based system.

6.3.7 Memory Comperision

To compare not only scalability and efficiency in terms of information dissemination, but also resource usage, Figure 6.12 illustrates the average RAM consumption per node relative to system size. A clear advantage of FogMon2 emerges: its RAM usage is approximately halved compared to DEMon, making it significantly more memory-efficient. This difference is primarily due to the underlying programming language and the tools used. FogMon2 is implemented in C++ and utilizes lightweight socket communication, whereas DEMon, as a prototype, is written in Python and relies on Flask-based HTTP servers, which are comparatively more resource-intensive per node. Despite these differences, both systems exhibit similar scalability, with resource overhead increasing only moderately as system size grows.

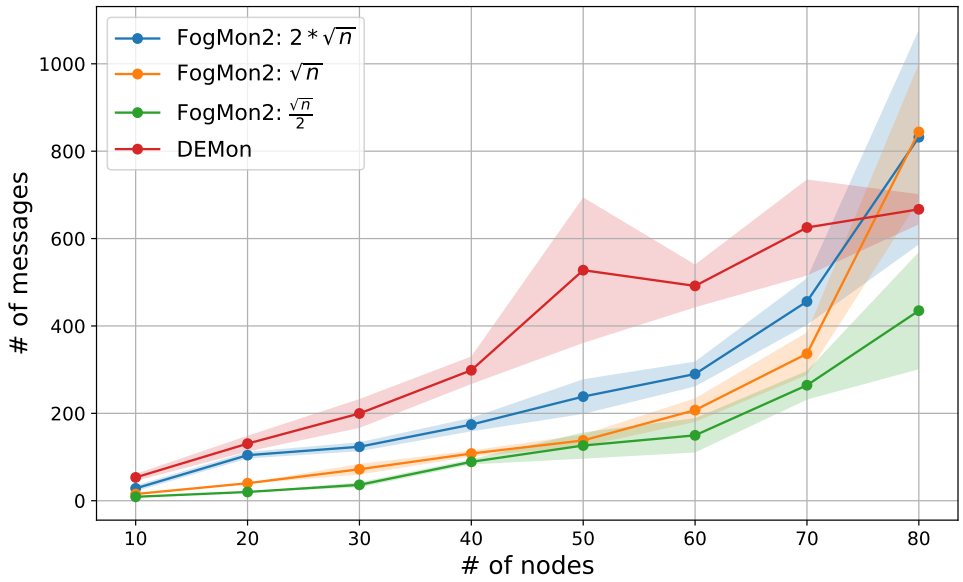


Figure 6.11: Number of messages till convergence with different FogMon2 settings compared to DEMon [IFTB24].

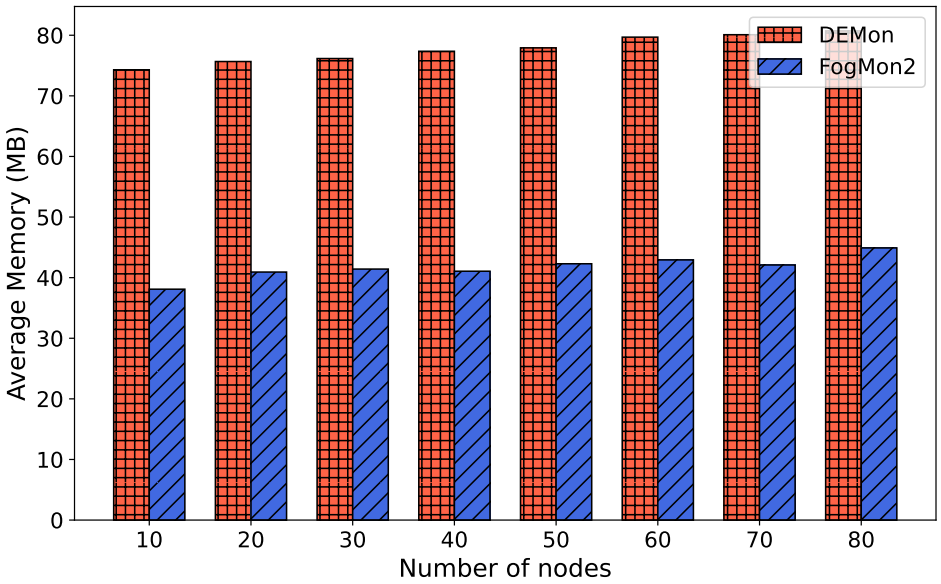


Figure 6.12: Memory Usage per node

6.4 Discussion of Findings

The evaluation results demonstrate that DEMon performs better than initially anticipated, particularly when compared to hierarchical monitoring systems like FogMon2. A key advantage of DEMon is its fully decentralized architecture, which eliminates the reliance on leader nodes and thereby avoids bottlenecks associated with hierarchical structures. The comparison revealed that DEMon achieves faster convergence in larger networks, where FogMon2's leader-based approach begins to introduce delays due to inter-leader communication overhead.

One of the major findings is that DEMon's hyperparameters, namely `gossip_count` and `gossip_rate`, provide effective mechanisms to adjust the speed of information propagation. The experiments showed that increasing `gossip_count` reduces the number of rounds required for convergence, demonstrating a trade-off between communication overhead and speed. Additionally, although `gossip_rate` influences the absolute convergence time, it does not affect the total number of messages required to reach system-wide synchronization, indicating that message complexity remains primarily governed by network size.

Regarding resource consumption, DEMon exhibits relatively low CPU and memory usage, even when considering the overhead introduced by its Python-based implementation. The system runs efficiently, with average CPU utilization below 1% and memory requirements remaining stable across different configurations. These findings suggest that the monitoring framework remains lightweight enough for deployment in resource-constrained edge environments.

A critical aspect of monitoring frameworks is their ability to ensure reliable data retrieval, even in failure scenarios. The evaluation confirmed that DEMon maintains robust query performance even under high failure rates. Thanks to its decentralized data storage approach, historical monitoring data remains available within the system, enabling accurate queries even when multiple nodes are unavailable. The Leaderless Quorum Consensus protocol ensures that queries return trustworthy results, making DEMon a reliable solution for distributed environments where node failures are common.

6.5 Limitations

While the evaluation demonstrated the strengths of DEMon, certain limitations must be acknowledged. One of the primary constraints is that all experiments were conducted in an emulated environment, meaning that real-world networking challenges, such as unpredictable latency variations and hardware failures, were not explicitly tested. While DEMon's robustness in failure scenarios was evaluated through controlled node disconnections, the absence of real-world deployment introduces uncertainties regarding its performance in dynamic edge environments. However, our previous work [IFTB24] has addressed this aspect by deploying DEMon in real-world settings, complementing the findings of this study.

Another limitation is the comparative analysis with FogMon2, which was only conducted for network sizes up to 80 nodes due to technical constraints in FogMon2's scalability. While this allows for a general performance comparison, it remains unclear how FogMon would behave in significantly larger networks compared to DEMon. Future work could explore additional decentralized monitoring solutions or extend the comparison with optimized FogMon2 configurations to assess long-term scalability differences.

DEMon's networking implementation relies on Docker's internal network infrastructure, which, while practical for an emulated testbed, could introduce bottlenecks when scaled on a single physical server. Unlike real distributed deployments where nodes operate across geographically distributed machines, Docker networking can introduce artificial constraints on communication latency and throughput. Investigating DEMon's performance in a fully distributed deployment across multiple physical hosts would provide a more comprehensive understanding of its scalability under real-world conditions.

From an implementation perspective, DEMon's Python-based architecture introduces additional computational overhead compared to lower-level implementations. While Python provides flexibility and rapid development, it consumes more CPU resources than optimized implementations in languages like C++ or Go. Additionally, DEMon relies on Flask and HTTP-based communication, which, while simplifying RESTful API integration, is less efficient than using direct TCP-based socket communication. A more lightweight messaging protocol could reduce overhead and improve performance, particularly in large-scale deployments.

CHAPTER 7

Conclusion and Outlook

This thesis introduced DEMon, a fully decentralized monitoring system designed for volatile edge environments. By removing the need for hierarchical aggregation, it enables efficient monitoring in distributed networks where centralized approaches would be impractical. Through gossip-based communication and peer-to-peer data retrieval, DEMon ensures system-wide information dissemination while maintaining low overhead. Its Leaderless Quorum Consensus (LQC) protocol further guarantees accurate data retrieval without reliance on fixed coordination points. The experimental evaluation demonstrated that DEMon scales effectively, maintaining stable performance even as system size increases. The transition from Kubernetes to Docker improved deployment flexibility and ensured a fully decentralized testbed. Compared to FogMon2, DEMon provided stronger scalability and resilience, though at the cost of higher resource consumption per node, largely due to its Python-based implementation and HTTP-based communication model. While the overall results validate the system's effectiveness, real-world deployment under heterogeneous conditions would further solidify its practical viability.

Certain challenges remain, particularly in optimizing resource usage and message efficiency. While DEMon effectively reduces redundant transmissions, it does not yet employ adaptive gossip mechanisms to dynamically adjust message frequency and dissemination targets based on network conditions. Additionally, CPU and memory usage could be further optimized by refining the implementation, possibly replacing HTTP-based communication with more lightweight socket-based messaging.

Future work could explore real-world deployments across diverse edge environments to assess performance under varying latencies, hardware constraints, and failure scenarios. Enhancing gossip strategies with adaptive mechanisms would further refine system efficiency, reducing message overhead while maintaining rapid convergence. Incorporating machine learning techniques for anomaly detection and predictive monitoring could also improve data accuracy and proactive resource management.

Overview of Generative AI Tools Used

In this thesis, OpenAI's ChatGPT-4o was used to enhance expression and grammatical correctness. All content, analyses, and scientific contributions are entirely my own. Additionally, DeepL was used as a translation tool to efficiently convert certain sections between German and English. To further refine linguistic accuracy, Grammarly was also employed to ensure additional grammatical precision.

Übersicht verwendeter Hilfsmittel

In dieser Arbeit wurde OpenAI's ChatGPT-4o verwendet, um Ausdrucksweise und grammatikalische Korrektheit zu verbessern. Sämtliche inhaltlichen Ausführungen, Analysen und wissenschaftlichen Beiträge stammen ausschließlich von mir. Zusätzlich wurde DeepL als Übersetzungstool genutzt, um bestimmte Abschnitte effizient zwischen Deutsch und Englisch zu übertragen. Zur weiteren sprachlichen Feinabstimmung kam zudem Grammarly zum Einsatz, um zusätzliche grammatikalische Präzision sicherzustellen.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABGM90] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, September 1990.

[Bak08] Lubomír Bakule. Decentralized control: An overview. *Annual Reviews in Control*, 32(1):87–98, 2008.

[BFG19] Antonio Brogi, Stefano Forti, and Marco Gaglianese. Measuring the fog, gently. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 523–538, Cham, 2019. Springer International Publishing.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[Bir07] Ken Birman. The promise, and limitations, of gossip protocols. 41(5):8–13, October 2007.

[BPM+18] Álvaro Brandón, María S. Pérez, Jesus Montes, Alberto Sanchez, and Marco D. Santambrogio. Fmone: A flexible monitoring solution at the edge. *Wirel. Commun. Mob. Comput.*, 2018, January 2018.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[CLC22] Jongwoo Choi, Il-Woo Lee, and Suk-Won Cha. Analysis of data errors in the solar photovoltaic monitoring system database: An overview of nationwide power plants in korea. *Renewable and Sustainable Energy Reviews*, 156:112007, 2022.

[CTCM22] Vera Colombo, Alessandro Tundo, Michele Ciavotta, and Leonardo Mariani. Towards self-adaptive peer-to-peer monitoring for fog environments. In *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '22, page 156–166, New York, NY, USA, 2022. Association for Computing Machinery.

[DC21] Chuan-Zhi Dong and F Necati Catbas. A review of computer vision–based structural health monitoring at local and global levels. *Structural Health Monitoring*, 20(2):692–743, 2021.

[Dou02] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[DQA04] Anwitaman Datta, Silvia Quarteroni, and Karl Aberer. Autonomous gossiping: A self-organizing epidemic algorithm for selective information dissemination in wireless mobile ad-hoc networks. In Mokrane Bouzeghoub, Carole Goble, Vipul Kashyap, and Stefano Spaccapietra, editors, *Semantics of a Networked World. Semantics for Grid Databases*, pages 126–143, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. *SIGPLAN Not.*, 25(3):1–10, February 1990.

[FGB21] Stefano Forti, Marco Gaglianese, and Antonio Brogi. Lightweight self-organising distributed monitoring of fog infrastructures. *Future Generation Computer Systems*, 114:605–618, 2021.

[FKMR09] Roy Friedman, Anne-Marie Kermarrec, Hugo Miranda, and Luís Rodrigues. *Gossip-Based Dissemination*, pages 169–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[FZTS11] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 73–84, 2011.

[GFPB23] M. Gaglianese, S. Forti, F. Paganelli, and A. Brogi. Assessing and enhancing a cloud-iot monitoring service over federated testbeds. *Future Generation Computer Systems*, 147:77–92, 2023.

[GI01] Samir Goel and Tomasz Imielinski. Prediction-based monitoring in sensor networks: taking lessons from mpeg. *SIGCOMM Comput. Commun. Rev.*, 31(5):82–98, October 2001.

[GK17] Marcel Großmann and Clemens Klug. Monitoring container services at the network edge. pages 130–133, 09 2017.

[GPG19] Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Gra86] James N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, 1986.

[IFDB22] Shashikant Ilager, Jakob Fahringer, Samuel Carlos de Lima Dias, and Ivona Brandic. Demon: Decentralized monitoring for highly volatile edge environments. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 145–150, 2022.

[IFTB24] Shashikant Ilager, Jakob Fahringer, Alessandro Tundo, and Ivona Brandić. A decentralized and self-adaptive approach for monitoring volatile edge environments, 2024.

[Jel11] Márk Jelasity. *Gossip*, pages 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, August 2005.

[JVM13] Stilian Stoev Joel Vaughan and George Michailidis. Network-wide statistical modeling, prediction, and monitoring of computer traffic. *Technometrics*, 55(1):79–93, 2013.

[JW08] Andrew D. Jurik and Alfred C. Weaver. Remote medical monitoring. *Computer*, 41(4):96–99, 2008.

[Jö13] Benjamin Jörissen. *Definition and Concepts of Monitoring and Evaluation.*, pages 99–100. 01 2013.

[KAH+19] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[Kel24] Brian Kelly. The impact of edge computing on real-time data processing. *International Journal of Computing and Engineering*, 5:44–58, 07 2024.

[KKH13] Anuj Kumar, Hiesik Kim, and Gerhard P. Hancke. Environmental monitoring systems: A review. *IEEE Sensors Journal*, 13(4):1329–1339, 2013.

[Kra18] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 8(8), 2018.

[LR15] Harald Lampesberger and Mariam Rady. *Monitoring of Client-Cloud Interaction*, pages 177–228. Springer International Publishing, Cham, 2015.

[LSKT17] Minh Le, Zheng Song, Young-Woo Kwon, and Eli Tilevich. Reliable and efficient mobile edge computing in highly dynamic and volatile environments. pages 113–120, 05 2017.

[LWW+22] Qianmu Li, Xudong Wang, Pengchuan Wang, Weibin Zhang, and Jie Yin. Farda: A fog-based anonymous reward data aggregation security scheme in smart buildings. *Building and Environment*, 225:109578, 2022.

[MH07] K. M. N. Muthiah and S. H. Huang. Overall throughput effectiveness (ote) metric for factory-level performance monitoring and bottleneck detection. *International Journal of Production Research*, 45(20):4753–4769, 2007.

[MMP11] Giuliano Mega, Alberto Montresor, and Gian Pietro Picco. Efficient dissemination in decentralized social networks. In *2011 IEEE International Conference on Peer-to-Peer Computing*, pages 338–347, 2011.

[MPL+22] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud continuum: The definition. *IEEE Access*, 10:131876–131886, 2022.

[MWO+21] Rodrigo Marino, Cristian Wisultschew, Andrés Otero, Jose M. Lanza-Gutierrez, Jorge Portilla, and Eduardo de la Torre. A machine-learning-based distributed system for fault diagnosis with scalable detection quality in industrial iot. *IEEE Internet of Things Journal*, 8(6):4339–4352, 2021.

[MXN+22] Jinhua Ma, Shengmin Xu, Jianting Ning, Xinyi Huang, and Robert H. Deng. Redactable blockchain in decentralized setting. *IEEE Transactions on Information Forensics and Security*, 17:1227–1242, 2022.

[NSV16] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. Nfvperf: Online performance monitoring and bottleneck detection for nfv. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 154–160, 2016.

[RBP11] Julien Rabatel, Sandra Bringay, and Pascal Poncelet. Anomaly detection in monitoring sensor data for preventive maintenance. *Expert Systems with Applications*, 38(6):7003–7015, 2011.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, page 329–350, Berlin, Heidelberg, 2001. Springer-Verlag.

[SGA+17] Hassan Jamil Syed, Abdullah Gani, Raja Wasim Ahmad, Muhammad Khurram Khan, and Abdelmuttlib Ibrahim Abdalla Ahmed. Cloud monitoring: A review, taxonomy, and open research issues. *Journal of Network and Computer Applications*, 98:11–26, 2017.

[Spl23] Splunk. What's it monitoring? it systems monitoring explained, 2023. Accessed: 2024-02-05.

[SPR05] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-Tolerant Networking*, WDTN '05, page 252–259, New York, NY, USA, 2005. Association for Computing Machinery.

[uA20] Mohammed Yousuf uddin and Sultan Ahmad. A review on edge to cloud: Paradigm shift from large data centers to small centers of data everywhere. In *2020 International Conference on Inventive Computation Technologies (ICICT)*, pages 318–322, 2020.

[UCR09] Deniz Uestebay, Rui Castro, and Michael Rabbat. Selective gossip. In *2009 3rd IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 61–64, 2009.

[VvRB03] Werner Vogels, Robbert van Renesse, and Ken Birman. The power of epidemics: robust communication for large-scale distributed systems. *SIGCOMM Comput. Commun. Rev.*, 33(1):131–135, January 2003.

[VWB+16] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26, 2016.

[ZWY+21] Zhili Zhou, Meimin Wang, Ching-Nung Yang, Zhangjie Fu, Xingming Sun, and Q.M. Jonathan Wu. Blockchain-based decentralized reputation system in e-commerce environment. *Future Generation Computer Systems*, 124:155–167, 2021.