RESEARCH-ARTICLE

# A Decentralized and Self-Adaptive Approach for Monitoring Volatile Edge Environments

**SHASHIKANT ILAGER**, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands

**JAKOB FAHRINGER**, Vienna University of Technology, Vienna, Vienna, Austria

**ALESSANDRO TUNDO**, Vienna University of Technology, Vienna, Vienna, Austria

**IVONA BRANDIĆ**, Vienna University of Technology, Vienna, Vienna, Austria

**Open Access Support** provided by:

**Vienna University of Technology**

**University of Amsterdam**



**Accepted:** 08 May 2025
**Revised:** 01 May 2025
**Received:** 30 September 2023

# A Decentralized and Self-Adaptive Approach for Monitoring Volatile Edge Environments

SHASHIKANT ILAGER, University of Amsterdam, The Netherlands
JAKOB FAHRINGER, TU Wien, Austria
ALESSANDRO TUNDO, TU Wien, Austria
IVONA BRANDIĆ, TU Wien, Austria

Edge computing provides resources for IoT workloads at the network edge. Monitoring systems are vital for efficiently managing resources and application workloads by collecting, storing, and providing relevant information about the state of the resources. However, traditional monitoring systems have a centralized architecture for both data plane and control plane, which increases latency, creates a failure bottleneck, and faces challenges in providing quick and trustworthy data in volatile edge environments, especially where infrastructures are often built upon failure-prone, unsophisticated computing and network resources. Thus, we propose DEMon, a decentralized, self-adaptive monitoring system for edge. DEMon leverages the stochastic gossip communication protocol at its core. It develops efficient protocols for information dissemination, communication, and retrieval, avoiding a single point of failure and ensuring fast and trustworthy data access. Its decentralized control enables self-adaptive management of monitoring parameters, addressing the trade-offs between the quality of service of monitoring and resource consumption. We implement the proposed system as a lightweight and portable container-based system and evaluate it through experiments. We also present a use case demonstrating its feasibility. The results show that DEMon efficiently disseminates and retrieves the monitoring information, addressing the challenges of edge monitoring.

Additional Key Words and Phrases: Edge Computing, IoT, Monitoring Systems, Self-Adaptive Systems, Gossip Protocol

## ACKNOWLEDGMENTS

This research was funded in part by the Austrian Science Fund (FWF) 10.55776/PAT1668223 and 10.55776/P36870; and by the Sustainable Watershed Management Through IoT-Driven Artificial Intelligence (SWAIN) project, CHIST-ERA-19-CES-005, Austrian Research Promotion Agency (FFG); and by Satellite-Based Monitoring of Livestock in the Alpine Region (Virtual Shepherd), FFG, and Austrian Space Applications Program (ASAP) #5307925.

## 1 INTRODUCTION

Edge computing offers computing resources for the latency-sensitive Internet of Things (IoT) workloads, enabling the data processing at the network edge [35, 38]. Unlike Cloud Computing, which provides reliable and robust computing resources from centralized data centers, edge computing offers services from a highly distributed environment with heterogeneous and resource-constrained compute and network resources. Despite its shortcomings, the massive demand for latency-sensitive and time-critical applications such as smart cities and autonomous

Authors' Contact Information: Shashikant Ilager, University of Amsterdam, Amsterdam, The Netherlands, s.s.ilager@uva.nl; Jakob Fahringer, TU Wien, Vienna, Austria, jakob.fahringer@tuwien.ac.at; Alessandro Tundo, TU Wien, Vienna, Austria, alessandro.tundo@tuwien.ac.at; Ivona Brandić, TU Wien, Vienna, Austria, ivona.brandic@tuwien.ac.at.


ACM 1556-4703/2025/6-ART
https://doi.org/10.1145/3736726

vehicles necessitates the widespread deployment of edge platforms [11, 36]. In this regard, deploying application services reliably on edge needs efficient infrastructure monitoring systems, allowing applications and service providers to make crucial decisions based on the monitoring data.

Monitoring services allow observation of the overall status of the infrastructure and play a crucial role in resource management tasks such as resource provisioning, scheduling, load balancing, and failure detection. Traditionally, cloud services are offered through multiple data centres directly managed by single service providers. Cloud data center resources are monitored through sophisticated Data Center Infrastructure Management tools centrally deployed on robust and reliable servers. Many of the cloud service providers build their in-house platforms (e.g., Google's Borgmon), while private clouds adopt open-source monitoring tools such as Zabbix [33] and Prometheus [42]. In all of such monitoring systems, both the *data plane* and *control plane*, possess centralized architecture. The *control plane* fine-tunes the configuration parameters such as data collection and forwarding intervals, parameters selection, and other application and infrastructure-specific configurations. A centralized system pushes the configuration rules to all the infrastructure nodes. On the other hand, the *data plane* is responsible for storing the monitoring data, i.e., the state of other servers. Such monitoring data is collected on predefined time intervals and usually stored in time-series databases hosted on centralized servers. However, such monitoring systems require dedicated and reliable computational and storage resources, which are infeasible at the edge due to their unique requirements and challenges.

*First,* edge Computing infrastructure is highly heterogeneous, consisting of IoT devices with onboard CPUs, embedded systems, domain-specific accelerators [34], and inexpensive off-the-shelf commodity servers and micro-data centres. Such hyper-heterogeneity with resource-constrained, failure-prone devices introduces a massive complexity to the design and implementation of monitoring systems, where a single configuration would not work for all types of nodes. *Secondly*, unlike a Cloud with a high-speed and reliable network, edge infrastructures are built upon limited bandwidth and unreliable networks, including wireless and cellular networks. Thus, communication failures should be considered a norm rather than an exception. *Thirdly*, edge infrastructures are volatile, where resources are pooled by multiple service providers across multiple network domains [11], and machines are dynamically provisioned or de-provisioned (join and leave the resource pool) based on network connectivity and power budget, among other parameters. This further challenges retrieving trustable monitoring data since resources belong to multiple parties [18, 46]. Therefore, we require a monitoring system that is completely decentralized for both *control* and *data plane*, where each participating node can control and fine-tune its parameters based on its capability and conditions. Despite some recent works have explored solutions for edge computing monitoring [9, 22, 40] and multi-tier Fog Computing [12, 19], they consider either some form of a centralized controller or remote storage mechanisms.

In this paper, we propose a *decentralized and self-adaptive monitoring system (DEMon)* for a highly-volatile edge environments. We envision the proposed monitoring system as distributed information management with efficient information spreading, storage, and data retrieval mechanisms. It provides a decentralized control and data storage method for edge monitoring system. We use a *stochastic group communication protocol* for information dissemination in a volatile edge environment [8, 25, 43].

DEMon leverages a *gossip-based information dissemination algorithm* capable of controlling according to the requirements of edge environments. The inherent stochastic characteristics of gossip-based protocol help create an adaptive and robust communication overlay network. This enables an effective decimation of information across the network in a decentralized manner without introducing massive concentrated network traffic on a specific network path and achieves a uniform network load distribution across the network.

In addition, we propose the *Leaderless Quorum Consensus (LQC)* protocol for information retrieval, which can quickly aggregate the information of a specific node, ensuring fast and trustworthy retrieval of the data. The DEMon's architecture is *decentralized*, that is, *it does not depend on a centralized controller and does not use centralized servers for information storage and retrieval*; instead, it uniformly distributes the data across

the network autonomously. Furthermore, it is *self-adaptive*, that is, no external configurations or measures are enforced during resource or network failures and infrastructure changes. Moreover, application services can timely access monitoring data without increasing latency like centralized monitoring systems.

This work extends our preliminary short paper [24] by: (i) providing a rigorous presentation of the monitoring system architecture; (ii) proposing a new state repository and network management techniques for efficient node insertion and deletion in the monitoring system; (iii) reporting results from an extensive empirical evaluation of the effectiveness of the approach and comparing it with a baseline; and (iv) showcasing the feasibility of the approach by simulating a real-world use case on a in-lab edge testbed.

In a nutshell, this paper provides the following contributions.

**A decentralized and self-adaptive monitoring system for highly-volatile edge environments.** We propose DEMon, a self-adaptive trustworthy monitoring system for highly-volatile edge environments that provides efficient and decentralized information spreading.

**A stochastic group communication protocol.** We present a gossip-based information dissemination algorithm and we study its hyper-parameters to understand their impact on edge environments monitoring.

**An efficient and trustworthy data retrieval method.** We propose the Leaderless Quorum Consensus (LQC) protocol for information retrieval, which can quickly aggregate trustable information of a specific node.

**A publicly available lightweight prototype.** We provide a publicly accessible prototype implementation of the DEMon monitoring system and algorithms.

**Empirical evidence of the effectiveness of the approach.** We answer four research questions by performing experiments in an emulated container-based environment. Results show that DEMon can efficiently spread and retrieve monitoring information across edge networks, in a scalable manner.

**An use case implementation of a mobile computing edge scenario.** We implement a mobile computing use case employing 12 RaspberryPi devices representing different edge nodes deployed in an urban area. Each node hosts both an edge AI-based application (i.e., object detection application) and the DEMon components. We execute the use case and collect feasibility evidence of the proposed monitoring system by using real data belonging to the edge User Allocation [27] dataset which has the recorded location entries of edge servers and mobile users.

The rest of the paper is organized as follows: Section 2 introduces the use case scenario motivating our solution approach and provides the required background details. Section 3 presents the architecture of DEMon, its components, and the main algorithms. Section 4 describes the implementation of the DEMon prototype. Section 5 presents the empirical evaluation and shows experimental results. Section 6 discusses the relevant related works. Finally, Section 7 presents concluding remarks and future directions.

## 2 MOTIVATION AND BACKGROUND

### 2.1 A Use Case for a Decentralized Edge Monitoring System

Let us introduce a mobile computing use case to illustrate the requirements of a decentralized and self-adaptive monitoring system for the Edge computing. Figure 1 depicts mobile users necessitating to offload their computational tasks (e.g., image object detection) to the nearby instances of an AI-based application. We consider that different resource provider dynamically pool their edge resources, and both the resource discovery protocol and the application instances are present on the edge nodes. Such application use cases using the edge nodes and AI-based applications can be found in many real-world applications such as environmental surveillance [1] for fire hazard detection and wildlife tracking, air and water quality monitoring [2], and smart traffic management

[1] https://sagecontinuum.org/docs/about/overview
[2] https://www.chistera.eu/projects/swain

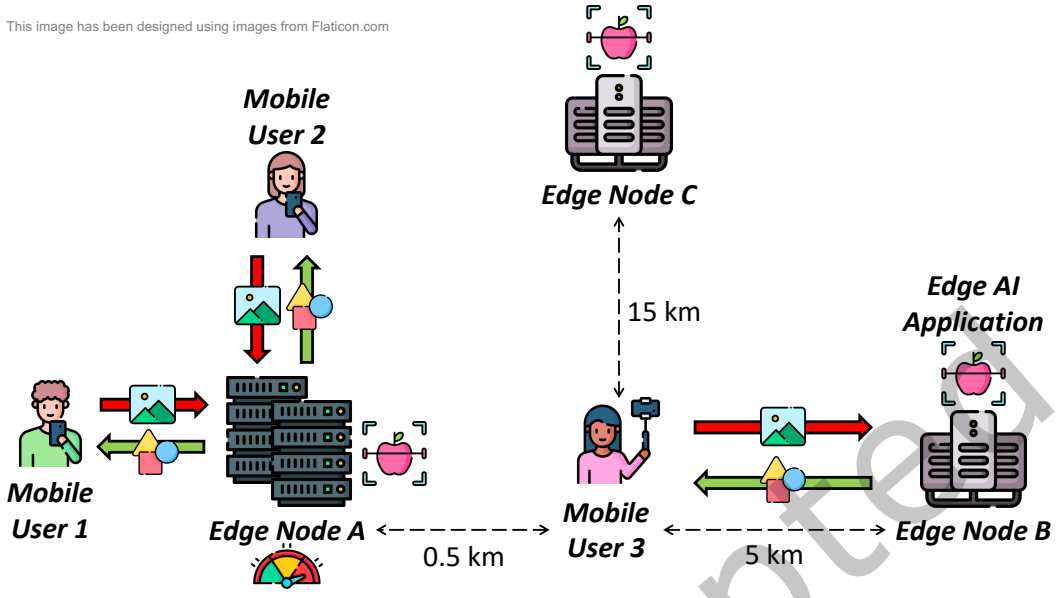



Fig. 1. An edge use case where mobile users offload a computational task to the nearest and less loaded edge node hosting an AI-based application.

systems [3], among others. Therefore, a decentralized monitoring system is necessary for efficient information dissemination, storage, and querying in resource-constrained and volatile edge environments.

Offloading task execution to edge nodes should be done with strict latency requirements (i.e., sub-milliseconds) to timely satisfy user needs and respect QoS requirements [23, 30]. For instance, to offload image processing to a nearby edge application instance, a mobile user has to find a suitable node that can accommodate their request by querying and aggregating information such as nodes availability, network and processing delay, and cost. In such cases, a centralized monitoring system is not feasible due to increased latency and failure-prone edge environments [12, 19].

In contrast, a mobile user can query the nearby edge nodes based on its current location, and obtain monitoring data that can help in finding a suitable node for their task. It must be considered this is possible only if the monitoring system is distributed, and the overall system's total or partial monitoring data is present in the nearby edge nodes. Moreover, providing trustable data is also required if we distribute the monitored data in an environment where multiple parties pool their resources to create a shared execution environment.

## 2.2 Gossip Protocol

The epistemic algorithms are prominently used in many networked systems [43] such as for distributed database replication, data aggregation, and data clustering [4, 15, 44]. Epistemic algorithms are inherently stochastic and provide a robust framework for building communication protocols and information systems [8, 43]. The *Gossip protocol* is a popular epistemic-based algorithm that provides efficient means for group communication without broadcasting. It is highly scalable and resilient, and it avoids a single point of failure [25, 45].

[3] https://intrasafed.ec.tuwien.ac.at

The Gossip protocol itself is straightforward. Periodically, each node randomly selects a few other nodes, exchanges the state information, and waits to receive data from other nodes. The protocol provides a framework to dynamically control communication efficiency and resource usage, while it manages trade-offs between them. For instance, the rate of message exchange (`gossip_rate`) and the number of random nodes chosen (`gossip_count`) are configurable parameters, which affect the overall performance of the protocol. Once a node receives state information, it updates its state based on whether or not the data is already present. If received data is already present or older than the current data (based on timestamp), it drops the message and waits for new messages. If all the nodes know about every other node, it is confirmed that the system is converged. However, in our case, since the monitored data also changes continuously at each node, the gossiping continues indefinitely based on monitoring time intervals.

It has been shown that the Gossip protocol works well in designing the theoretical distributed systems [8]. However, its application in large-scale real-world systems is less explored [16]. In addition, if the hyper-parameters of the protocol, such as `gossip_rate` and `gossip_count` are misconfigured, it might exponentially increase the network and storage load, and it may even perform worse than broadcast-based communication [1]. Therefore, it becomes imperative to carefully study the feasibility of this stochastic method and analyze its effect on resource consumption, especially in resource-constrained edge environments. Consequently, in this paper, we study the effect of various hyper-parameters on metrics such as network and storage usage, speed of information propagation, query latency, and information quality.

## 3 DEMON: DECENTRALISED EDGE MONITORING AND CONTROL

The DEMon monitoring system is designed following four main principles:

(1) **Self-adaptation and configuration.** A system is self-adaptive and configurable if it can control and operate autonomously under dynamic conditions [26]. In that regard, DEMon does not depend on other nodes for control instructions and has no bootstrap configurations when new nodes join the system. Each node only sends and receives monitoring information through an independent DEMon agent that runs autonomously. The hyper-parameters can be configured, suiting system properties such as network and application workload.
(2) **Trustworthiness.** A system is trustworthy if it provides accurate data without compromising with the data security [32, 37]. In particular, it is more challenging to provide trustworthiness when the data is stored distributed, and replicated across a decentralized network of nodes. DEMon relies on a leaderless *quorum-based consensus* (LQC) protocol to provide a trustworthy data retrieval mechanism.
(3) **Robustness.** A system is robust when it is able to operate despite the occurrence of one or more failures [10, 44]. DEMon provides robustness by using a decentralized architecture for its control and data plane. Stochastic information dissemination helps to balance the network load, and it avoids a single point of failure that may bring to complete data loss. Moreover, information about a node can be always retrieved, even if that node is currently unavailable since the information of all nodes is probabilistically stored in a decentralized fashion.
(4) **Efficient communication and data retrieval.** The DEMon communication protocol optimizes network efficiency during data dissemination and retrieval. Its information dissemination Gossip protocol is designed to avoid unnecessary data transmission, and the querying LQC protocol provides a latency-sensitive response in run-time.

Figure 2 depicts the architecture of the DEMon monitoring system. The system functionalities are realized within four main components: (i) State Repository, (ii) Information Dissemination Controller, (iii) Information Retrieval Controller, and (iv) DEMon Query Client.

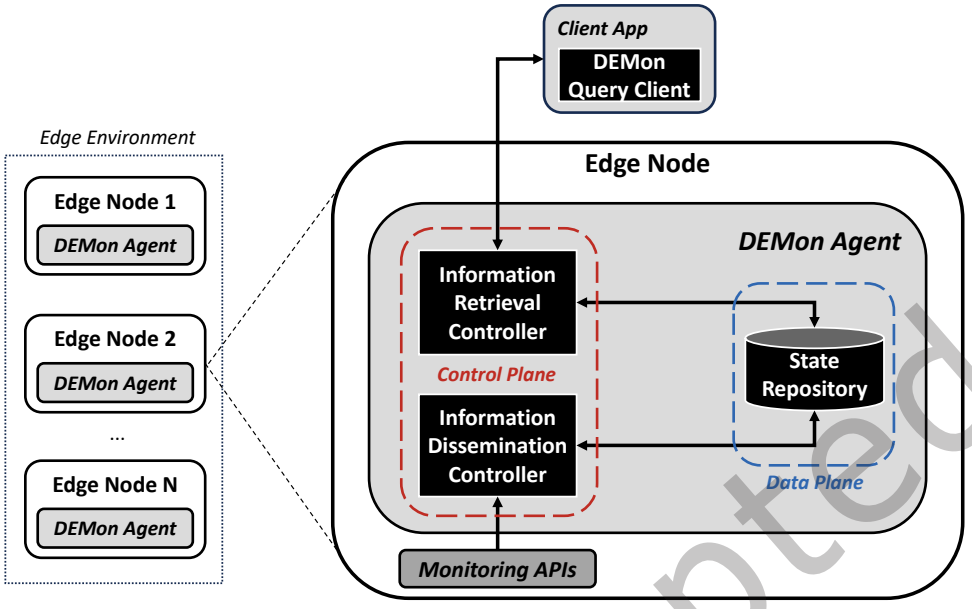



Fig. 2. The architecture of the DEMon monitoring system.

The *State Repository (SR)* contains the node states, that is, it contains both the monitoring data from the node hosting the repository and monitoring data belonging to some or all of the other nodes participating in the network.

The *Information Dissemination Controller (IDC)* is responsible for information dissemination, that is, it both sends and receives monitoring data leveraging a gossip-based algorithm. In particular, it collects monitoring data monitoring APIs (e.g., OS-specific or application APIs), and it stores them within the *SR*. Then, it periodically sends the current system state, which includes its own state and the state of other nodes currently present in the *SR*. Simultaneously, it waits asynchronously for the monitoring data sent by other nodes, and it only save the latest data to the *SR*.

The *Information Retrieval Controller (IRC)* is responsible for information retrieval, that is, it provides an Application Program Interface (API) to retrieve monitoring data for any incoming request from client applications. It scans the *SR* and returns back the requested information.

The *DEMon Query Client (DQC)* is used by client applications that want to query the DEMon monitoring system and retrieve monitoring data. It provides a trustworthy data verification mechanism by leveraging the proposed Leaderless Quorum Consensus (LQC) protocol.

Here, both the IDC and the IRC represent our control plane, and SR represents the data plane. Please note that each node running the DEMon Agent (DA) executes both controllers, and they are managed independently of each other. Each node also possesses its own SR, resulting in a homogeneous network of nodes with the same role and providing the same set of functionalities. Thus, our system provides a completely decentralized architecture for both the control plane and the data plane (SR). On the other hand, centralized monitoring systems like Prometheus [42] or Zabbix [33] possess a central controller and data plane, making them unsuitable for volatile edge environments.

In the following subsections, we describe each of the DEMon components in detail.

### 3.1 State Repository (SR)

Listing 1. Example of a State Repository entry.

```
{
    "192.168.167.100": [
        {
            "metrics": {
                "cpu": 35,
                "memory": 2048,
                "storage": 10,
                "network_latency": 4,
                "heartbeat": 1695131028
            },
            "counter": 2,
            "unreachable_by": ["192.168.167.102"],
            "digest": "6ec3e0143db76fc0fbdd9d00311230fe4a1cfb565b73909fd9286a23f5e168a1"
        },
        {
            ...
        }
    ]
}
```

The State Repository is a repository containing an historical record of node states. In particular, it both contains the states of the node hosting the repository, plus the states of some or all the nodes participating the network. Formally, $SR$ is a set of node states $\{S_1, \ldots, S_n\}$, where $S_i$ is the set of states for node $i$ identified by an ID, e.g., its IP address. We define $S$ as a time ordered set of tuples $\{s_1, \ldots, s_p\}$, where $s_t$ is the node state at time $t$. The node state $s$ is a tuple $(M, c, U, d)$, where $M$ is a set of metric values $\{m_1, \ldots, m_k\}$ with $m_j$ representing the value of metric $j$ (e.g., CPU consumption); $c$ is an incremental counter that represents the number of gossip rounds; $U$ is the set node IDs that were unable to reach a node when selected for a gossip round; and $d$ is the resulting hash value (digest) obtained by computing the hash function $hash(M, c, U)$.

Listing 1 shows a concrete example of entry of the SR for a node identified by its IP address. The mapping with the tuple $S$ is straightforward: `metrics` represents the set of metrics $M$, `counter` represents the counter $c$, `unreachable_by` represents the set of node IDs $U$, and `digest` represents the hash value $d$.

### 3.2 Information Dissemination Controller (IDC)

The IDC is responsible for disseminating information (i.e., monitoring data) across the network of nodes leveraging a gossip-based protocol for an efficient data distribution. In particular, it provides two main functionalities: (i) it periodically collects monitoring data from its own execution environment (e.g., resource consumption or application-specific metrics), stores the collected values to the *SR*, and disseminates the monitoring data to a subset of nodes participating the network; (ii) it asynchronously waits for incoming monitoring data from the other nodes, and it stores only the latest to the *SR*. Each node can opportunistically configure the controller configuration parameters, i.e., time interval for data sending (`gossip_rate`) and number of messages (`gossip_count`), enabling control of the compute and network resource consumption at run-time, and information convergence speed.

To exemplify the two IDC functionalities, we illustrate the dissemination logic using the pseudo-code in Algorithms 1 and 2, respectively. From now on we use the term *sending node* when referring to a node executing Algorithm 1, and *receiving node* when referring to a node executing Algorithm 2. However, please note that both

the algorithms run in all the edge nodes, guiding the logic for sending and receiving the monitoring data of each node.

**Algorithm 1:** Information Dissemination Controller - Send Node States

```
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
```

Algorithm 1 shows the logic executed by a *sending node*, which gossips monitoring data to the other nodes of the network. Here, for every time interval defined by the `gossip_rate` parameter, the IDC first computes the new node state $s$ by gathering monitoring data (line 2), and it updates the $SR$ (line 3). Then, it *randomly* chooses a subset of nodes equal to the configured `gossip_count` parameter from the $SR$ (line 4) to start gossiping. If a selected node has been already declared unreachable by a predefined configurable number of distinct nodes (i.e., `failures_count`) (line 6), then such node is considered as failed and deleted by the $SR$ (line 7). More details about the deletion logic are provided in Section 3.2.1.

Since a sender can choose random nodes to gossip, it is necessary to avoid duplicate or unnecessary data communication. If a receiver node already has the same or the most recent data as the other nodes, the sender should avoid gossiping such data to the receiver. To address this, initially, the *sending node* only sends its node state $s$ and metadata of its current known information stored in the $SR$ (line 9). Metadata includes all the known node IDs and their corresponding counters ($c$). This logic avoids excessive bandwidth consumption by excluding sending the duplicate data repetitively between the nodes.

It is also important to note that, a *receiving node* receiving the gossip message only intends to obtain missing or up-to-date data from other nodes. Consequently, the *receiving node* replies back with a list of node IDs whose up-to-date state is requested (`requests`), and all the node states (`updates`) that it has fresher state $s$ than the

sending node (i.e., because of the counters present in the metadata) (line 12). The *sending node* then stores new node states within its *SR*, and also sends back the requested node states to the *receiving node* (lines 13-17). Therefore, by choosing to only communicate the metadata first, and then sending the actual requested data, DEMon avoids excessive resource consumption of edge nodes. Finally, if there is no response, the *receiving node* is marked as unreachable (line 19).

**Algorithm 2:** Information Dissemination Controller - Receive Node States

```
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
```

Algorithm 2 shows the logic executed by a *receiving node*, which receives a gossip message from the other nodes in the network. For any incoming message, it first updates the *SR* with the sender node state (line 3). Then, it checks if its *SR* needs to be updated based on the metadata it has received, i.e., node IDs and counters (lines 4-10). Based on the comparison, the *receiving node* sends back a response message composed by two parts: (i) node IDs requiring fresher states (`requests`), (ii) up-to-date node states that the *sending node* does not have yet (`updates`). Thus, the *sending node* also receives new node states from the *receiving node* within the same communication cycle. In fact, the *sending node* is simultaneously disseminating new information across the network, but also getting up-to-date data (if any) from the *receiving node* in single gossip round. This two-way communication enables quick information sharing between nodes, enabling efficient and faster information dissemination.

The stochastic selection of nodes in Algorithm 1 ensures the uniform distribution of messages in the network, and it does not spike a specific network link, increasing bandwidth usage. Moreover, the information spreads exponentially across the nodes and evenly distributes the network load [1]. Any permanent or transient failures will not affect the monitoring infrastructure. New nodes can send and receive monitoring data from the network and quickly know about all the nodes.

*3.2.1 Nodes Network Management.* Volatile edge environments are characterized by a dynamic network topology, with nodes joining and leaving at any time unpredictably, thus, enabling dynamic scalability for resource optimization [5, 29]. To this extent, efficiently managing nodes' membership in an edge network is crucial, as the system size (i.e., the number of nodes) can change rapidly. However, this complicates the nodes' network management regarding new node insertion and deletion of a failed node in the monitoring system, leveraging a gossip-based communication protocol.

***Node Insertion.*** DEMon assumes any node can join the edge monitoring network without any extra network configuration: they just need to know the identifier (e.g., IP address) of at least one of the nodes to start gossiping.

In fact, when a new node with an unknown identifier joins the system and sends its state to a node already participating in the monitoring network, the latter inserts the new node state $s$ in its $SR$ according to the IDC logic explained above. This step allows the new node to participate in the gossip-based communication protocol, exchange data with other nodes, and contribute to the overall monitoring process.

***Node Deletion.*** On the other hand, a node can also disconnect from the network for various reasons, such as resource or network failures, or even voluntary disjoining. Therefore, it is important to detect and delete a disconnected node from the $SR$ to reduce the data storage and increase the gossip-based protocol reliability (i.e., avoiding choosing a disconnected node for a gossip round). However, distinguishing between a node that is experiencing transient and temporary failures (e.g., short network disconnection), and a node that failed is a challenging problem [3]. If a node is deleted too aggressively, i.e., because of not receiving a single response, information re-convergence could take several rounds. Hence, a safe and reliable deletion is a key functionality of a decentralized monitoring system.

The DEMon embeds the node deletion strategy within the IDC logic (See Algorithm 1). When a *sending node* does not receive a response from a *receiving node* during a gossip round, it marks such node as unreachable (i.e., it adds its own identifier $ID$ to the set $U$, denoting that it identified a potential failed node). This information is also gossiped in the following round as part of the node state $s$. When the size of the $U$ set of a node reaches the `failures_threshold` (e.g., 3), that is, $k = failures\_threshold$ distinct nodes were unable to gossip to such node, DEMon considers it as failed or disconnected, and deletes the node from the $SR$. This ensures that DEMon does not delete any node for a transient failure or temporary network issues.

The node deletion strategy introduces reliability through distributed observation of failures and synchronization of this information across all nodes without any central coordination. This logic also affects communication and storage efficiency, as it stops the propagation and retention of such failed node data within the $SR$. By discontinuing the gossiping to a failed node and saving data storage, DEMon optimizes its resource allocation, ensuring that resources are not wasted on processing unreliable information. Furthermore, the decisively failed status is communicated throughout the network as part of the usual IDC, in the subsequent gossip rounds, enabling other nodes to discover failure simultaneously.

Nevertheless, when a failed node engages in gossiping and receives a valid response from another node, it signifies a restoration of connectivity, resulting in the reset of its decisive failed status. This adaptive mechanism acknowledges the importance of allowing nodes to recover from temporary failures and participate actively in the monitoring process, ensuring that network reliability is upheld. Please note that DEMon assumes the presence of security overlay mechanisms such as node authentication and authorization, which is out of the scope of this work.

*3.2.2 State Repository Management.* The $SR$ can be represented as a key-value data structure where the `key` of each entry is the node identifier $i$ (e.g., its IP address), and the `value` is the historical set of node states $S_i$. Such data structure can be persisted in memory, allowing fast read/write operations. For an incoming gossip message, the IDC checks if it contains any new state for a node, comparing the current counter $c$ to the one received, and updating the $SR$ if needed. Thus, the $SR$ size (i.e., number of keys) remains constant at any time after the system converges unless new nodes join the system, allowing greater scalability.

However, the amount of memory required to store node states exponentially grows because at each gossip round a new node state $s$ is added to the node states set $S$, potentially creating memory issues on resource-constrained edge devices. To this extent, DEMon only stores the recent $r$ gossip rounds data of all nodes in-memory, and checkpoints at regular intervals to centralized data storage, enabling offline processing and analysis if required. For example, in every $r$ elapsed gossip rounds, all the previous $r - 1$ node states are persisted to the centralized storage, and the in-memory $SR$ is refreshed retaining only the latest node states. The checkpoint interval can be

either static or dynamically adaptable based on the gossip round, and each node can independently govern its own checkpoint interval.

Checkpoints coming from different nodes have probabilistically duplicated node states, resulting in an exponential increase in the checkpointing data size. To further optimize the checkpointing logic to avoid duplicated storage, we store a single entry for each edge node in the checkpointing data storage, and only new data received from other nodes is additionally persisted (i.e., based on its counter values in *SR*). This ensures that the centralized storage maintains an up-to-date representation of the node states without excessive data redundancy.

These design choices optimize the utilization of resources within the edge environment. In fact, by keeping the most relevant information in memory, nodes can efficiently engage in real-time monitoring and data sharing. Periodic checkpointing ensures that a historical record of information is maintained, allowing comprehensive monitoring of the system designed for both online and offline batch processing.

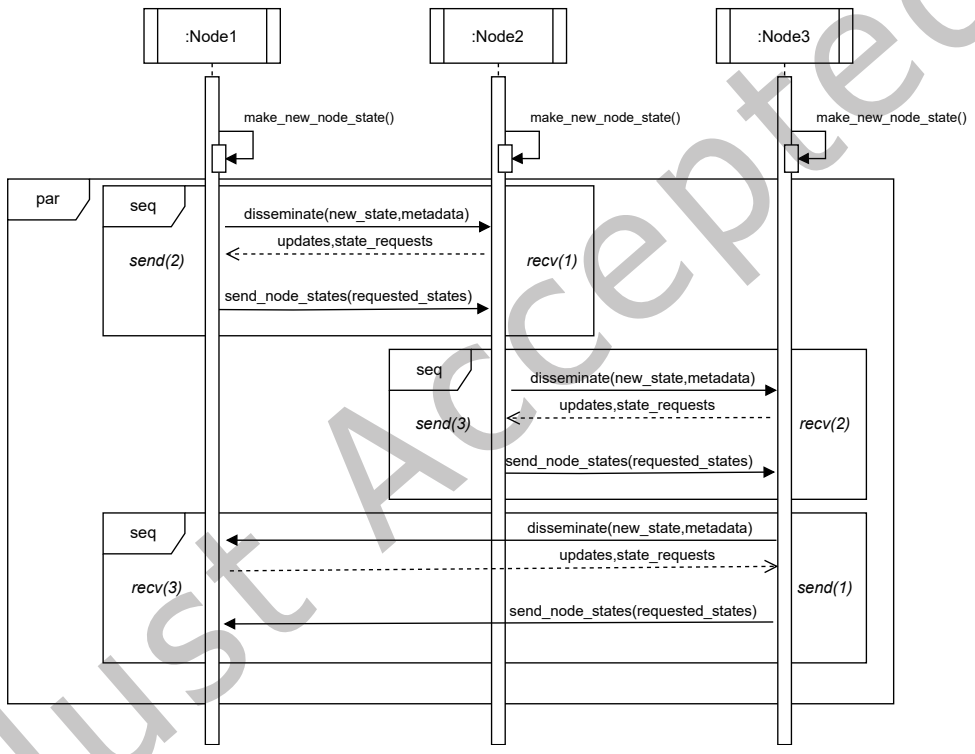



Fig. 3. The sequence diagram of the DEMon monitoring system illustrates the flow of messages between nodes in one round of gossiping.

*3.2.3 Illustration and Running Example of the IDC.* Figure 3 presents a sequence diagram that illustrates the sequence of messages exchanged among nodes during a gossip round as part of the Information Dissemination Controller (IDC). To simplify, we depict three nodes engaged in gossiping with `gossip_count` set to 1, exchanging monitoring information. Although the diagram shows a sequential message exchange, it is important to note that all nodes gossip concurrently. For example, Node 1 initiates a gossip send message to Node 2 through random selection (*send(2)*) with its current metadata (node IDs and counters) from *SR*. Upon receiving the message

(*recv(2)*), Node 2 compares the metadata with its *SR* and sends back any updated data it has, as well as requests for any data missing from its *SR*. Subsequently, Node 1 sends the requested data to Node 2, completing Node 1's gossip cycle for that round. It is crucial to note that, in parallel, Node 1 is also receiving incoming gossip messages from other nodes during the same round. For instance, while Node 1 is communicating with Node 2, Node 3 sends a gossip message to Node 1, and Node 2 sends a gossip message to Node 3. This parallel communication accelerates information dissemination. The number of messages that are sent in a gossip round can be controlled with the `gossip_count` parameter, further speeding up the information dissemination per gossip round.

Table 1. Illustration of state management with possible DEMon runtime operations (system size of 5). Notations R: Round; A: Action(s); SR: State Repository (with $[SR_1] \rightarrow [SR_2]$: $SR_1$ = begin of round & $SR_2$ = end of round); U: unreachable_by (with (x,y) → z: z is unreachable by x & y).

| R | | Node 1 | Node 2 | Node 3 | Node 4 | Node 5 |
|---|---|---|---|---|---|---|
| 1 | A | *send(2), recv(3)* | *send(3), recv(1)* | *send(1), recv(2), recv(4)* | *send(3),accept(5)* | *membership(4)* |
| | SR | [1]→[1:3] | [2]→[1:3] | [3]→[1:4] | [4] →[3:5] | [5]→[4,5] |
| | U | [ ] | [ ] | [ ] | [ ] | [ ] |
| 2 | A | *send(3)* | *send(3)* | *send(4), recv(1), recv(3)* | **Internal Error** | *send(4)* |
| | SR | [1:3]→[1:4] | [1:3]→[1:4] | [1:4]→[1:4] | - | [4,5]→[4,5] |
| | U | [ ] | [ ] | (3) → 4 | - | (5) → 4 |
| 3 | A | *send(5)* | *send(4), recv(3)* | *send(2), recv(5)* | Internal Error | *send(3), recv(1)* |
| | SR | [1:4]→[1:5] | [1:4]→[1:4] | [1:4]→[1:5] | - | [4,5]→[1:5] |
| | U | (5)→4 | (2,3) → 4 | (3,5) → 4 | - | (3,5) → 4 |
| 4 | A | *send(4), recv(2)* | *send(1), recv(5)* | *send(5)* | Internal Error | *send(2), recv(3)* |
| | SR | [1:5]→[1:3,5] | [1:4]→[1:3,5] | [1:5]→[1:5] | - | [1:5]→[1:3,5] |
| | U | (1:3,5) → 4 | (2,3,5) → 4 | (3,5) → 4 | - | (2,3,5) → 4 |

We further illustrate the IDC process with an example depicted in Table 1. This table provides a high-level overview of the state information *SR*, for each node at the start and end of a gossip round. For simplicity, we consider an edge system comprising 5 nodes with a `gossip_count` of 1. We define a gossip round as *R*and actions as *A*. Within the state repository *SR*, we display only the node ID, represented as an integer. The list $U$, denoting `unreachable_by`, contains the IDs of nodes unresponsive to gossip messages. It is important to note that $U \in SR$, but for clarity, we present it separately. We consider that *SR* & $U$ is updated at the end of each gossip round and *send()* & *recv()* only propagate the *SR* and $U$ with the updated state of the previous round.

Initially, Nodes 1-4 are part of the network, each possessing its own monitoring knowledge in their *SR*. In round *R*1, Node 5, not yet present in any *SR*, announces its network membership by initiating the gossip process. It randomly selects Node 4 (*send(4)*) and transmits a gossip message containing its metadata from *SR*. Concurrently, Node 4 receives the message, acknowledges Node 5's membership, and incorporates Node 5's ID into its *SR*. Meanwhile, Nodes 1-3 continue to *send()* and *recv()* messages as depicted in Figure 3, with Node 4 also disseminating its *SR* to Node 3.

During round *R*2, Node 4 fails, and Nodes 3 and 5 attempts to gossip (*send(4)*) to it. The absence of a response leads them to add Node 4's ID to their respective $U$ lists, which will be propagated in the subsequent round.

In round *R*3, Node 2 sends a gossip message to Node 4 and, receiving no reply, adds Node 4's ID to its $U$. Simultaneously, Node 3 updates its *SR* after receiving a message from Node 5.

Finally, in round *R*4, Node 1 attempts to communicate with Node 4 and, encountering no response, adds Node 4's ID to its $U$. Node 1 also updates its *SR* based on a message from Node 2. At this stage, at Nodes 1, 2, and 5 $failures\_threshold$ reaches 3 (from list $U$) of Node 4, prompting the dissemination of its failed state across the

network. Consequently, Nodes 1, 2, and 5 remove Node 4's ID from their *SR*, as the *U* count for Node 4 hits 3. Additionally, the system achieves convergence, with all active nodes aware of each other. This process continues, ensuring that information regarding new and failure nodes is circulated autonomously as part of the gossiping monitoring information in the network.

### 3.3 Information Retrieval Controller(IRC)

The IRC provides an API to access the *SR* and retrieve node states. Its functionality is straightforward, that is, it scans the *SR* looking for the requested node IDs, and returns the corresponding node state sets $S_1, \ldots, S_n$. Currently, the IRC does not provide any query language to filter or sort data, it only accesses node states by using the node identifier. However, a more sophisticated query language can be incorporated into the IRC to enable enhanced query capabilities.

### 3.4 DEMon Query Client (DQC)

A critical aspect of DEMon is providing an efficient logic to obtain the monitoring information of an edge node in near real-time. Since data is replicated across the participant edge nodes, it is crucial to ensure that retrieved data is recent, consistent, and trustworthy. Generally, consistency is ensured in distributed information systems using the consensus protocols like Paxos [28]. However, such protocols require the election of leader nodes, which creates a centralized bottleneck and increases latency, making them impractical in edge environments.

To this extent, we propose an efficient *Leaderless Quorum Consensus (LQC)* protocol, where we eliminate the need for centralized leader nodes architecture. We introduce a leaderless consensus method where a client querying for the monitoring information reaches to consensus with decentralized participants. Such leaderless protocols have found applications in many recent distributed database systems such as Cassandra, [4] allowing faster information retrieval. The protocol is implemented by DQC, which pseudo-code is shown in Algorithm 3.

**Algorithm 3:** DEMon Query Client implementing the Leaderless Quorum Consensus Protocol

```
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
```

When the DQC has to perform a query to the DEMon system, it spawns parallel queries to a random subset of nodes (i.e., predefined `quorum`), and obtains responses consisting of the queried data, and the associated message digests (lines 1-3) from the IRC. Once it receives the minimum number of responses (i.e., `quorum`), it initially compares their counters (line 4). If the counter of responses matches, then corresponding digests are matched.

[4] https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-15%3A+General+Purpose+Transactions

If this condition holds, then the data is considered consistent and trustworthy. Otherwise, the current request session is discarded, and a new set of nodes is chosen randomly (lines 5-11).

Please note that the QC does not provide any feature for retrieving granular monitoring information because it just relies on the IRC API to obtain the node states. Rather, it provides the ability to retrieve data of a node or aggregated data of all nodes in a decentralized manner.

### 3.5 Summary of Methodological Innovations and Contributions

The main contributions and methodological innovations of DEMon are summarized as follows:

- **A fully decentralized and robust monitoring system:** DEMon introduces a monitoring system for highly volatile edge environments using a gossip-based protocol resilient to node failures. While gossip-based protocols facilitate efficient dissemination of information across the network, their stochastic nature often leads to information duplication. To mitigate this, we employ a time counter and metadata exchange protocol before sending new data in our IDC data. Additionally, we propose exchanging fresh data as part of the receiver acknowledgment message, further accelerating overall data transfer speed. These enhancements, tailored to the requirements of the monitoring system, are novel to gossip-based protocols.
- **Trustworthiness and weak consistency of the IRC:** DEMon's IRC protocol ensures significant performance advantages while providing weak consistency. It also offers the reliability required for volatile edge environments by avoiding a single point of failure. DEMon IRC incorporates a deterministic and irreversible digest encoded in the node state, ensuring data integrity (i.e., monitoring information is not altered by a malicious node) for any query result.
- **Investigation of various hyper-parameters:** DEMon examines the impact of gossip hyper-parameters on network and storage usage, speed of information propagation, query latency, and information quality. Although theoretical analyses of gossip-based protocols exist, their practical application in distributed networked systems and overhead analysis is less explored, which is crucial in resource-constrained edge environments. Misconfiguration can severely affect the network and storage load, leading to worse performance than broadcast-based communication. Accordingly, in this work, we provide a comprehensive analysis of hyper-parameters and their impact on performance indicators.

Trustworthiness can be even further improved by including mechanisms such as digital signatures, certificates, reputation management systems, and proof-of-work systems. However, such techniques significantly increase the computational and storage cost, thus, they are not advisable considering the resource constraints that characterize edge environments.

### 3.6 Theoretical Complexity and Analysis

The primary complexity of our gossip-based monitoring system arises from the IDC sender algorithm (Algorithm 1). Since the receiver algorithm (Algorithm 2) accounts for asynchronous messages, its asymptotic complexity is always $O(1)$.

Therefore, we describe the theoretical time complexity of the IDC sender algorithm (Algorithm 1). The time complexity of this algorithm is $O(n \cdot m)$, where $n$ is the number of nodes selected for gossiping and $m$ is the number of updates received in response, such that $0 \leq m < n$. If our system is highly reliable and does not experience any failures, then $n = m$.

The outer loop runs at a fixed gossip rate, which does not affect the asymptotic complexity. Inside the loop, most operations (such as `make_new_node_state()`, `store_node_state()`, and `select_nodes_to_gossip()`) are assumed to be $O(1)$. The main complexity arises from the inner conditional loop that iterates over the selected nodes. For each node, there is a constant time check and potential deletion. The dissemination and response parsing are assumed to be $O(1)$. The condition that iterates over updates has a complexity of $O(m)$, where $m$ is

the number of received updates. Additionally, getting and sending requested node states is $O(1)$. Since this inner loop runs for each selected node, the overall complexity becomes $O(n \cdot m)$.

It is important to note that in a typical gossip protocol, the time complexity (information convergence) is $O(\log N \cdot m)$, where $N$ is the total number of nodes in the network [15, 25]. Here, $0 \leq m < n$, since we assume we would have $m$ responses to $n$ messages sent. The theoretical gossip protocol complexity analyzes the convergence time, i.e., how many gossip rounds it takes to disseminate information across the network completely. In our case, even after convergence, the gossip continues since the monitoring information continuously generates new data. Thus, for any new state of the data received at one node, the time complexity of one gossip round bounded by $n$ gossip selected nodes ($1 \leq n < N$) is $O(n \cdot m)$. At the same time, the overall convergence of the current state is $O(\log N \cdot m)$.

The space complexity of this algorithm is primarily determined by the size of the SR, which is the total number of nodes in the network and the number of timestamps from history we choose to store. Since this timestamp threshold is always constant, the space complexity is $O(N)$ in the worst case.

## 4 PROTOTYPE IMPLEMENTATION

We implemented the DEMon monitoring system in a prototype available at https://github.com/hpc-tuwien/DEMon/releases/tag/acm-taas-submission. Both the *DEMon Agent (DA)* components and the *DEMon Query Client (DQC)* are implemented using Python language.

The *DA* exposes HTTP JSON APIs leveraging the Flask framework[5] to implement the *IDC* and *IRC* functionalities. In particular, the *IDC* registers an HTTP endpoint to handle incoming gossip messages, while the *IRC* registers an HTTP endpoint to handle queries performed by the *DQC*. Moreover, the *IDC* spawns a dedicated thread to collect monitoring data (i.e., CPU and memory consumption, disk usage, and network I/O) using the *psutil tool*[6] and send gossip messages. The *SR* is implemented as an in-memory Python dictionary, while the checkpoint's database is realized by an SQLite[7] instance.

The *DQC* is provided by a Python module that can be imported into third-party client applications that want to retrieve trustworthy monitoring data using the LQC protocol.

Finally, we provide a Dockerfile to build a Docker image and start the DEMon agent as a containerized application.

## 5 PERFORMANCE EVALUATION

This section describes how we conducted the performance evaluation and discusses the experimental results. In particular, we investigate the following four Research Questions (RQs) to evaluate our approach.

- $RQ_1$: **Convergence.** ***How quickly can DEMon discover all the nodes in the edge network and how do different parameters affect the convergence speed?*** This RQ investigates how quickly each node knows about every other node participating in the network (i.e., convergence). Also measures the time required for a new node to know about the other nodes when it joins the network. It is important to note that each node will update its monitoring information continuously, and the gossiping continues forever to spread the recent information across the system.
- $RQ_2$: **Storage and Network.** ***What is the storage and network overhead of the DEMon in the information dissemination process?*** This RQ investigates the overhead introduced by DEMon in terms of of storage and communication, and it studies whether the system could scale with different numbers of nodes on the edge.

[5] https://flask.palletsprojects.com/en/2.2.x/
[6] https://pypi.org/project/psutil/
[7] https://www.sqlite.org/index.html

RQ$_3$: **Query Latency.** ***How many messages does it take to find the requested information using the proposed LQC protocol?*** This RQ investigates the number of messages required to retrieve the information about any particular node based on Algorithm 3.

RQ$_4$: **Age of Information (AoI).** ***How do we measure the quality of the information retrieved from DEMon?*** Since the monitoring information is continuously updated, it is essential to know the timeliness of the information stored in each node. In our case, it is the freshness of the data in each node. To this end, we measure the Age of Information (`AoI`) metric [2]. The `AoI` of any node in the system is defined as:

$$AoI = \frac{1}{n}\sum_{i=1}^{n} t_i - u(t_i) \tag{1}$$

where $n$ is the total number of other nodes' information a node has, $t_i$ is the actual time counter of remote $node_i$, and $u(t_i)$ is the current time counter of $node_i$ locally. To understand the `AoI`, let us assume the system has two nodes, `node_1`, and `node_2`, and the monitoring system has reached an initial convergence state (every node knows everyone). The `AoI` of `node_1` is the difference between the time counter for `node_2` at `node_1` i.e., $u(t)$ and the actual time counter at `node_2` i.e., $t$. In our case, the time counter is an integer value (as described in Section 3.1); each node increments its time counter in a predefined time interval.

## 5.1 Baseline

We select FogMon2 [19, 21] as the baseline to compare our approach. FogMon2 is a hierarchical peer-to-peer (P2P) monitoring system for Fog-Edge environments. Despite there are other relevant edge monitoring solutions available in the literature, such as PyMon [22] or FMonE [9], FogMon2 is the most suitable for a comparison with a fully decentralized system as DEMon.

FogMon2 leverages a hierarchical two-layer architecture composed of Leader and Follower nodes. Followers forward monitoring data to their Leader at regular intervals (i.e., report time), while Leaders share monitoring data with each other in a P2P fashion that is similar to a gossip-based algorithm. Among many configuration parameters available, the number of Leaders with respect to the size of the overall system is the most influential factor for the performance and robustness of FogMon2.

We executed FogMon2 using Docker containers. On our testbed server, we tested with a system size of up to 80 nodes, which was the maximum possible on our testbed server.

In particular, we experiment with multiple leader node sizes (i.e., $2 \times \sqrt{N}$, $\sqrt{N}$, and $\frac{\sqrt{N}}{2}$), where $n$ is the total number of nodes in the system. The number of Follower nodes is initially evenly distributed among all the Leaders and then balanced by FogMon2 itself.

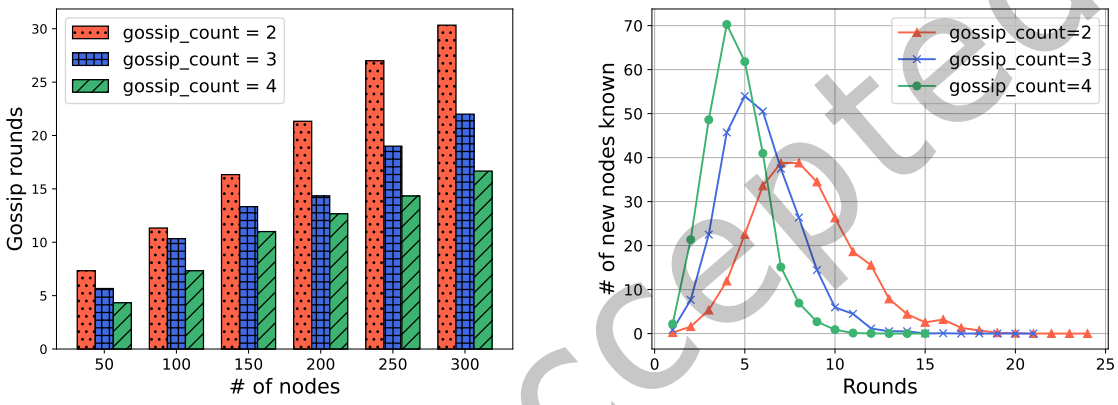
## 5.2 Experimental Setup

We evaluate DEMon on a Docker-based testbed to create a large-scale realistic edge environment, feasible for repetitive and inexpensive experiments. We execute Docker containers on a bare-metal server (40-core Intel(R) Xeon(R) CPU E5-2630L, 128GB RAM) hosted in our HPC laboratory. Each container executes the DEMon Agent and represents an edge node in our setup. We scaled edge nodes (i.e., containers) from 0 to 300, with an interval of 50. Please note that no changes are required to deploy our monitoring system on the new edge infrastructure, either physical or virtualized, provided the nodes can run as Docker containers.

Our monitoring framework's performance depends on multiple hyper-parameters including `gossip_count`, `gossip_rate`, `failure_rate`, and the system size $n$. Thus, we conducted experiments with different hyper-parameter values and analyzed the results. The details of these hyper-parameters are presented in the Table 2.

Table 2. The hyper-parameters used in the experiments, their definition and configuration values

| Parameter | Defintion | Values |
|---|---|---|
| gossip_count (integer) | Defines how many target nodes are selected for each gossip round | {2, 3, 4} |
| gossip_rate (seconds) | Defines the time interval between two gossip rounds | {1, 5, 10, 15, 20} |
| failure_rate (%) | Defines % of failed nodes | {10, 20, 30, 40, 50 60, 70, 80, 90} |
| n: system size (integer) | Defines number of nodes in the edge environment | {50, 100, 150, 200, 250, 300} |




(a) Convergence vs rounds (gossip rate = 1).

(b) Gossip rounds vs known nodes (system size=300, gossip rate=1). The higher the gossip count, the faster the information spreads.

Fig. 4. Performance of gossip protocol.

All the experimental material to reproduce the experiments is available in our publicly released repository[8].

## 5.3 Results and Analysis

*5.3.1 $RQ_1$: Convergence.* In the following experiments, we scaled the number of edge nodes, $n$ (i.e., containers) up to 300 with an interval of 50. All experiments are repeated three times, and average values are considered for the analysis.

Figure 4a shows an analysis of the number of gossip rounds required to reach the initial convergence state. Since each node has its gossip round running locally, at any instant, we consider the maximum gossip round among all nodes as system convergence (when all nodes know about all other nodes). Here, we configured `gossip_count` = $\{2, 3, 4\}$ and `gossip_rate` to 1. The higher value of `gossip_count` spreads the information quickly, and the system converges faster in lesser gossip rounds as observed in Figure 4a. This behavior denotes that when a new node (re)joins the system, it is able to know about all other nodes in fewer gossip rounds. For instance, when gossip `gossip_count` is set to 4, the system converges within four rounds ($n$ = 50). Similarly,

[8]https://github.com/hpc-tuwien/DEMon/releases/tag/acm-taas-submission

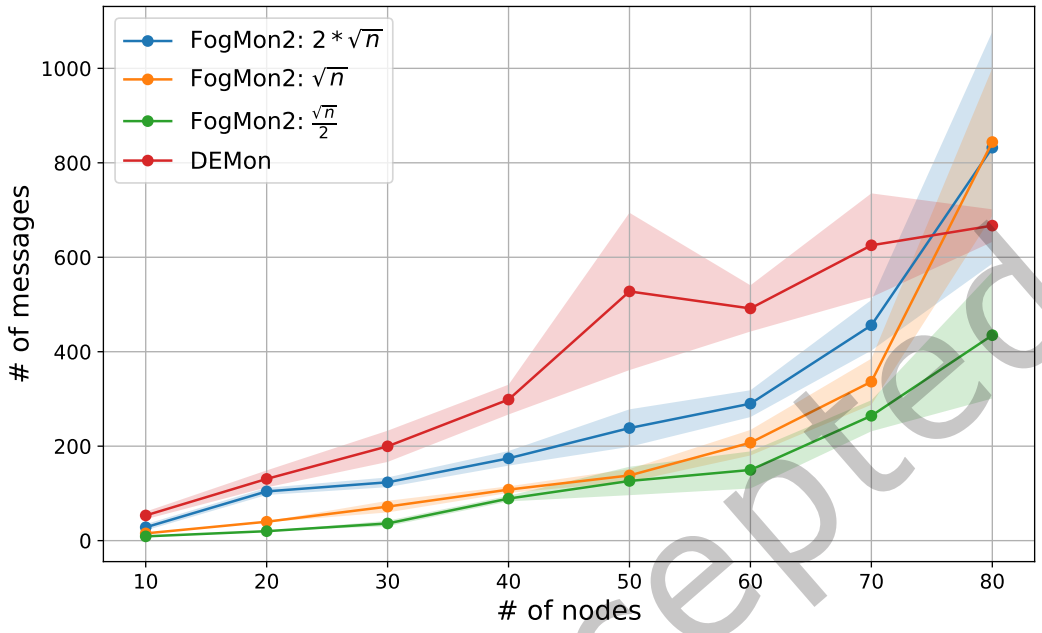



Fig. 5. A number of messages for convergence between FogMon2 and DEMon. Here, we set 3 different configurations of FogMon2 (number of Leader nodes) and gossip_count=2 for DEMon.

Figure 4b shows the number of new nodes known (average value of all nodes) during each gossip round. As observed, more than 50% of nodes' data is gathered within the first four gossip rounds ( `gossip_count` = 4).

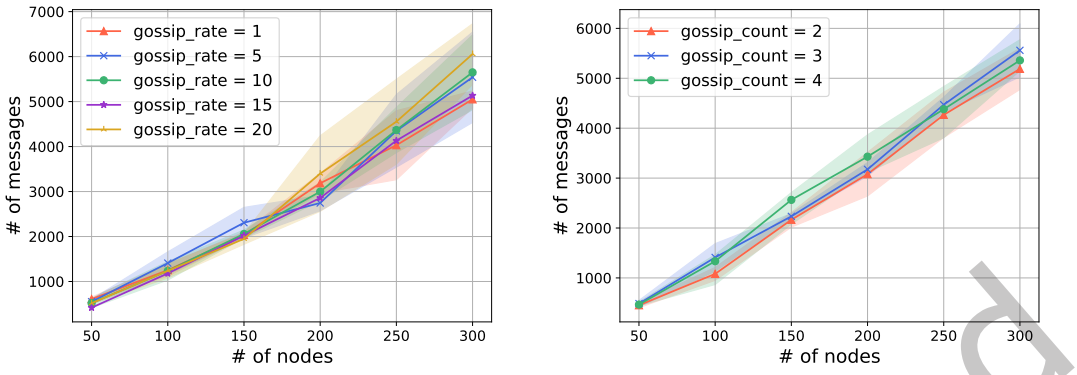
**Comparison with Baseline:**

Figure 5 presents the performance of DEMon in comparison to FogMon2 [21]. We modified the existing FogMon2 [9] tool to collect the number of messages the Leader nodes send and receive until convergence is achieved.

The efficiency and convergence speed of FogMon2 can be influenced by the number of Leader nodes, which disseminate the monitoring information among themselves. We consider the system to be converged when all Leader nodes are aware of all participating nodes in the network. We tested three Leader node configurations as recommended by FogMon2: $2 \times \sqrt{N}$, $\sqrt{N}$, and $\frac{\sqrt{N}}{2}$. We used the number of messages required for convergence as a performance indicator.

As depicted in Figure 5, DEMon initially requires more messages due to its completely decentralized communication architecture, unlike FogMon2. However, as the system size grows, the gap in the number of messages between the optimal configuration of FogMon2 $\sqrt{N}$ and DEMon narrows. With a larger system size, DEMon and FogMon2 have comparable performance in terms of number of messages required for convergence, demonstrating DEMon's capability of being a completely decentralized system and performing similarly to partially decentralized approaches like FogMon2.

[9] https://github.com/hpc-tuwien/DEMon/tree/major_revision




(a) Effect of gossip rate on number of messages (gossip count = 3).

(b) Effect of gossip count on number of messages (gossip rate = 1).

Fig. 6. Sensitivity analysis: Effect of gossip rate and gossip count parameter on the number of messages for convergence with different numbers of nodes.



**Hyper Parameter Analysis.**

Figure 5.3.1 depicts the effects of `gossip_rate` and `gossip_count` on the number of messages during the initial system convergence. Figure 6a shows the number of messages it takes to converge the system with different `gossip_rate`. Here, the plots also depict the average value from three different runs and their standard deviations. Each node's monitoring interval and `gossip_rate` are set to similar; thus, for each `gossip_rate` second, a node sends its new state. We configured `gossip_rate` = {1, 5, 10, 15, 20} seconds and `gossip_count` to 3. When the number of nodes increases, the total number of messages simultaneously increases, demonstrating that a higher number of message exchanges are required for many nodes, as shown in Figure 6a. Irrespective of `gossip_rate`, the number of messages is similar across all node sizes; representing convergence requires an almost equal number of messages for fixed system size and does not depend on `gossip_rate`.

Similarly, `gossip_count` also significantly affects the performance of the monitoring system. The `gossip_count` decides how many random nodes are chosen for each gossip round. Figure 6b depicts the effects of `gossip_count` on the number of messages. We configured `gossip_count` = {2, 3, 4} and `gossip_rate` is set to 3 in these experiments. Similar to `gossip_rate`, number of messages do not vary significantly across different system size (see Figure 6b).

Although `gossip_rate` and `gossip_count` do not affect the number of messages it takes to converge the system, they significantly affect the convergence time. Figure 7 shows the effect of these parameters on time. The x-axis in Figure 7 shows different `gossip_rate`, and grouped bar plots represent `gossip_count` configurations. As observed, the `gossip_rate` directly affects the convergence time since it controls the number of messages per second.

Therefore, if we want to control the bandwidth usage or speed of convergence, `gossip_rate` can be configured accordingly. The smaller `gossip_rate` results in faster convergence time and vice versa. Similar observations can be drawn for the `gossip_count` parameter. Thus, if faster convergence and a higher rate of information exchange are required, `gossip_count` should be set to a higher value, and `gossip_rate` should be set to a smaller value. If minimal bandwidth consumption is necessary, contrast values can be set to these parameters.

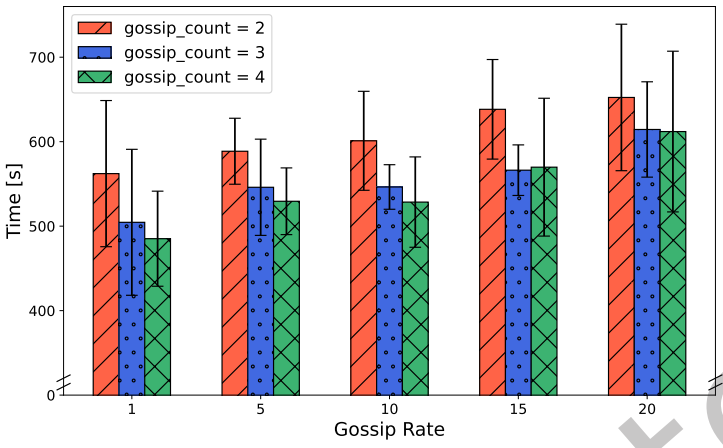
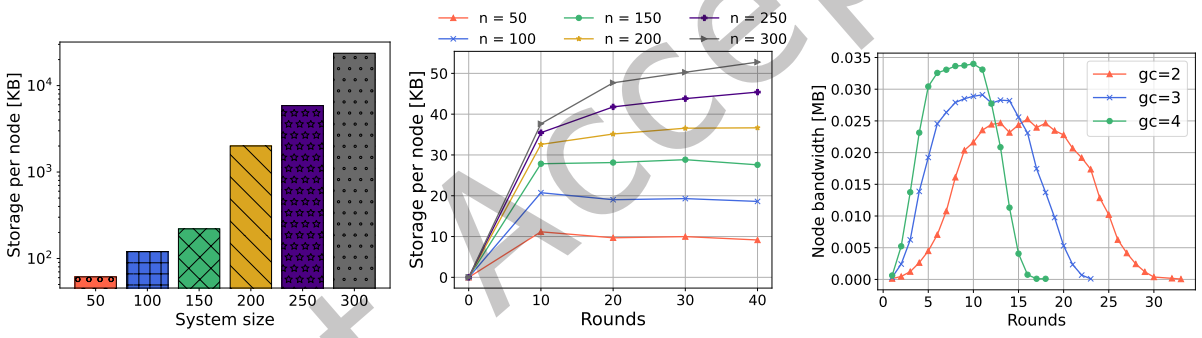



Fig. 7. Sensitivity analysis: Effect of gossip rate and gossip count parameter on the time for convergence with different numbers of nodes (n = 300).




(a) Storage consumption after convergence (gossip_count = 3).

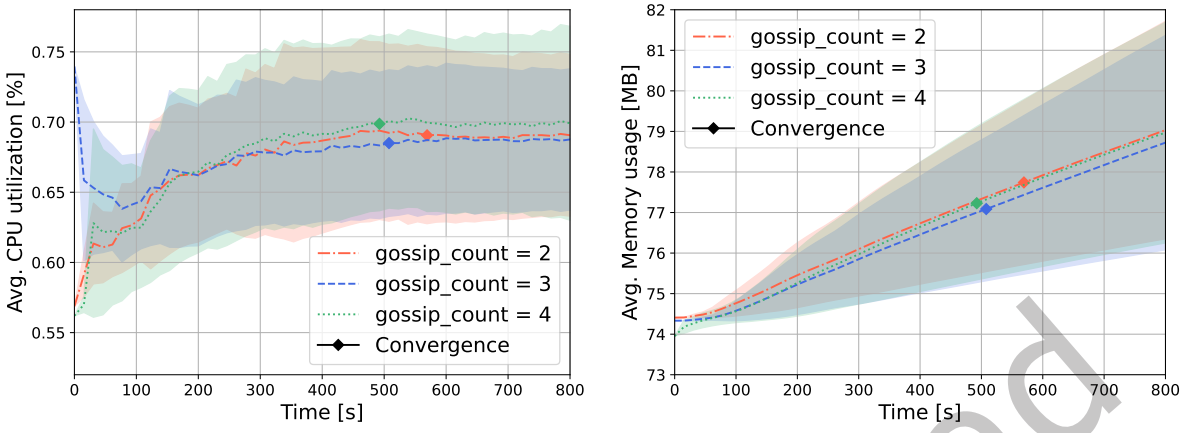
(b) Storage consumption every 10th round (gossip_count = 3).

(c) Bandwidth till convergence (n = 300, gc = gossip_count).

Fig. 8. Analysis total storage after convergence, storage consumption over gossip rounds and bandwidth consumption over gossip rounds till convergence (gossip rate = 1).

**Answer to RQ1:** DEMon significantly enhances the convergence rate of monitoring information (e.g., less than 25 rounds when n=300, `gossip_count` =3), offering increased control over both the speed of convergence and resource utilization (number of messages). This is achieved through the fine-tuning of configuration parameters—specifically `gossip_rate` and `gossip_count`, which can be adjusted to meet the specific demands of the edge environment.

*5.3.2 $RQ_2$: Resource Usage Analysis.* To evaluate the storage and network overhead of DEMon we measure the size of the state repository *SR* at each node until convergence and after convergence, and also the bandwidth consumption across gossip rounds.




(a) CPU Usage

(b) Memory Usage

Fig. 9. Analysis of Resource Usage by DEMon (system size=300, gossip_rate=1).

The average results across all nodes are presented in Figure 8. Figure 8a shows the average size of *SR* with different system sizes when the overall system has converged. For instance, with a maximum system size of 300, the state repository has 23.68 MB (23698 KB) in its size, demonstrating the lightweight memory footprint of DEMon. Similarly, Figure 8b shows the size of *SR* across different gossip rounds. The storage size increases exponentially until the system converges since new nodes' states are added to *SR*. After that, we have a linear increase and almost a constant size in storage size. It is important to note that, we refresh the in-memory *SR* for every 10th round, to keep the most recent run-time data in the system. The average network bandwidth across different gossip rounds can be seen in Figure 8c. The network bandwidth has a similar trend as storage: it is high until convergence since a large amount of data needs to be gossiped initially. After the system converges, DEMon consumes extremely low levels of bandwidth. These empirical results of storage and network consumption show that DEMon can scale with low resource footprints on the edge resources.

We also analyze the CPU and memory resource consumption of DEMon agents. Figure 9 illustrates the CPU usage (Figure 9a) and memory usage (Figure 9b) of a DEMon agent. For the CPU usage analysis, we maintained a system size of 300, with a gossip rate set to 1. The figure presents the average resource consumption across all 300 edge nodes (containers). For each data point, we calculate the cumulative average and display the cumulative standard deviation. The marked points on the scatter plot indicate the time at which the monitoring system has converged (when each node knows every other node). As observed, our monitoring agent consumes less than 0.75% of CPU resources, demonstrating DEMon's lightweight design and effectiveness in resource-constrained environments. The CPU consumption exhibits a consistent pattern even post-convergence, indicating stable CPU resource utilization.

Similarly, Figure 9b depicts the cumulative average memory consumption of all the edge nodes. Notably, until convergence, the memory consumption increases as new nodes' data is incorporated into *SR*. Post-convergence, the memory consumption shows a decline. This behavior can be modulated based on the number of timestamps of past data retained in our system (currently, we store all historical data in memory to show the maximum possible usage). In conclusion, DEMon's resource utilization is suitable for resource-constrained edge environments due to its minimal CPU and memory footprints.

Furthermore, we examined FogMon2's CPU and memory consumption with a system size of 80 (the maximum possible on our server) and found that FogMon2, on average, exhibits a CPU usage of 5.9343% and a memory usage of 42,217 MB across leaders and followers. The higher CPU usage of FogMon is due to the fact that it integrates resource-intensive monitoring probes (e.g., bandwidth testing), whereas we utilize the lightweight "psutil". A direct comparison of resource usage between DEMon and FogMon2 is not accurate since both tools employ different probes. DEMon collects limited metrics with a lightweight tool, and our focus is more on efficient communication overlay, storage, and retrieval. Existing probes could be embedded for more accurate measurements and a rich set of monitoring metrics.

**Answer to RQ2:** DEMon exhibits a significantly lower overhead in resource consumption, making it highly suitable for resource-constrained edge environments. For example, it maintains an average CPU usage of less than 0.75%. Additionally, the node bandwidth significantly decreases once the system converges. DEMon also has a smaller memory footprint, with the flexibility to adjust the runtime memory size according to specific requirements.

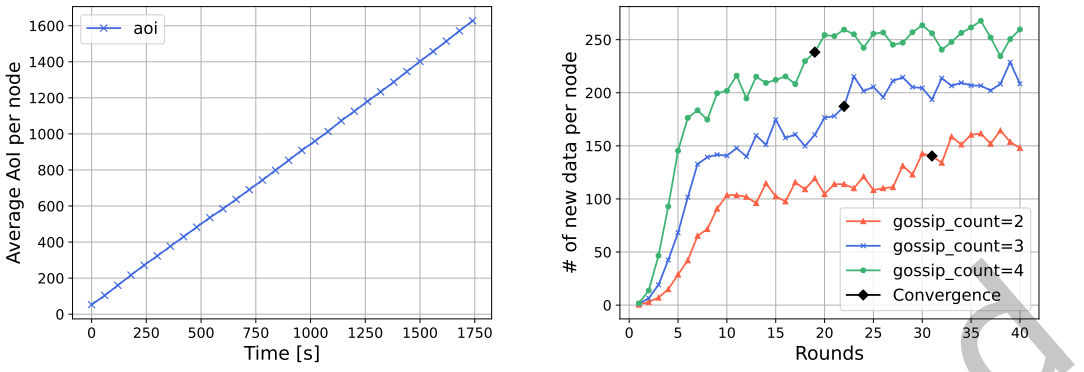
*5.3.3 $RQ_3$: Query Latency.* To investigates the number of messages required to retrieve monitoring data about any particular node, we performed 100 queries each time for different failure rates. In this setting, each query requests the utilization metrics of a random node. The querying process follows the logic explained in the Algorithm 3, where `quorum_number` is set to 3, similar to popular distributed databases such as Cassandra.

We generate parallel queries to three random nodes for each query request. Once a node receives a request, it checks the local object store based on the requested node ID (i.e., IP address) as key and sends the monitoring data in the format described earlier (see Section 3.1). We set different failure rates from 0%-90% with the interval of 10%. We randomly disconnect the `failure_rate` % of existing nodes for each failure rate setting. Despite this, DEMon can remarkably provide the information of a requested node without query failures. This is because, since every node stores information of every other node, even when 90% of the nodes fail, the queried node information is successfully retrieved. In our results, we received the queried information with a maximum number of messages of 148 and a minimum number of messages of 3 (best case scenario, equal to `quorum_number`), with an average value of 10.45 and a median of 3.

**Answer to RQ3:** DEMon offers a resilient framework capable of handling a high number of node failures. It ensures reliable data retrieval from decentralized nodes even in scenarios where up to 90% of the nodes have failed.

*5.3.4 $RQ_4$: Age of Information (AoI).* Although DEMon provides monitoring information of all nodes even when a high % of nodes fail, it is essential to know the freshness of the data in all nodes, which is also a valuable aspect of monitoring data. We analyze the average `AoI` of all nodes for different time intervals. In this experiment, we take a snapshot of the monitoring data from all nodes every minute for a total of 30-minute intervals. Once the system is converged initially, the `AoI` of each node is calculated based on Equation 1.

Figure 10a shows the overall `AoI` of the system (average from all the nodes) for all intervals. It can be observed that `AoI` of the overall monitoring system gradually increases as the monitoring time interval increases. This is expected since every node gossips to a very few nodes for every `gossip_rate` seconds; it would take several further gossip rounds to reach the updated information to other nodes, while each node simultaneously keeps on updating their monitoring information, creating a time delay between the two data instances. However, as Figure 10b shows, even after initial system convergence, all the monitoring agents receive other nodes' newly updated monitoring information in each round. The rate of new updates is highly dependent on the `gossip_count`. When `gossip_rate` is set to 4, the average number of new fresh data updates is almost equal to the system size (i.e., 300). Since DEMon is designed to hold multiple snapshots of a node's data in different nodes. If the most recent or fresh data is needed, higher `gossip_count` values can be configured.




(a) Age of Information analysis (gossip_count = 3).

(b) New updated data of other nodes received (black diamond marker denotes the initial system convergence).

Fig. 10. Analysis of AoI and new data updates (system size=300, gossip_rate=1).

**Answer to RQ4:** The results demonstrate that all edge nodes consistently receive new updates from their peers. Furthermore, DEMon offers controllable parameters that allow for the management of data freshness throughout the edge network.

## 5.4 Use Case Evaluation on an RPi-based Testbed

In this section, we showcase the implementation of the use case described in Section 2.1. Our primary objective is to provide fast and trustworthy monitoring information for near real-time mobile user applications deployed on top of volatile edge nodes.

Figure 11 depicts our Raspberry Pi based testbed, with 3 interconnected mini edge clusters (ECs), where each EC has 4 Raspberry Pi (RPi) nodes, for a total of 12 edge nodes. We also setup an additional RPi controller node to run our experiments. All the experimental material to reproduce the use-case is available in our publicly released repository[10].

We emulate the geographical location of edge nodes and users based on the Edge User Allocation dataset [27]. This dataset provides the geo-locations of edge servers and users in the metropolitan area of Melbourne, Australia. We derive 12 edge server locations and associate their latitude and longitude values with our RPi nodes. Similarly, we generate the user mobility behaviour using the user location data from the same dataset, which has the location entries of users based on their mobile GPS location. We deploy the DEMOn Agent as a Docker container to all RPi nodes.

We use an object detection AI model as a representative workload in our IoT-Edge environment. We use a pre-trained YOLO object detection model [11] trained with the COCO dataset and deploy it on our Raspberry Pi cluster. The application takes a base64 encoded image as the input and returns a list of detected objects and bounding box dimensions of the objects as a response. We implement this application with a Python-based user application and deploy it as a Docker container. An instance of this application runs inside each edge node and exposes its services through HTTP APIs.

[10]https://github.com/hpc-tuwien/DEMon/releases/tag/acm-taas-submission
[11]https://pjreddie.com/darknet/yolo/

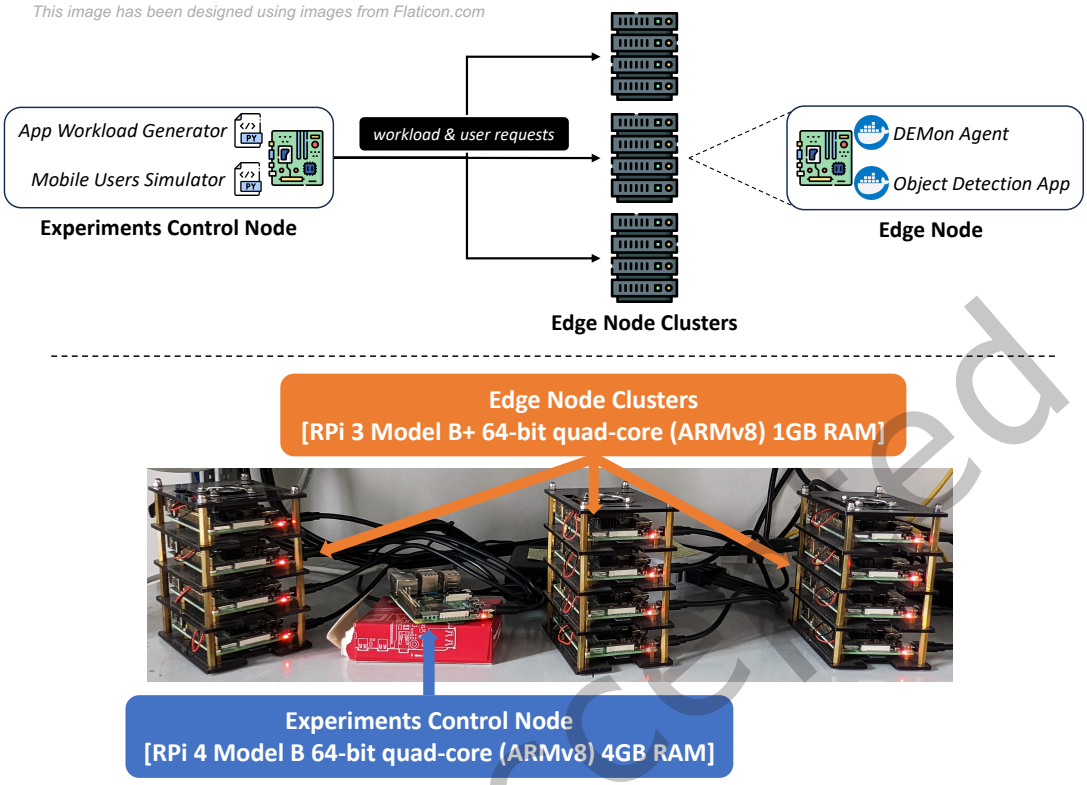


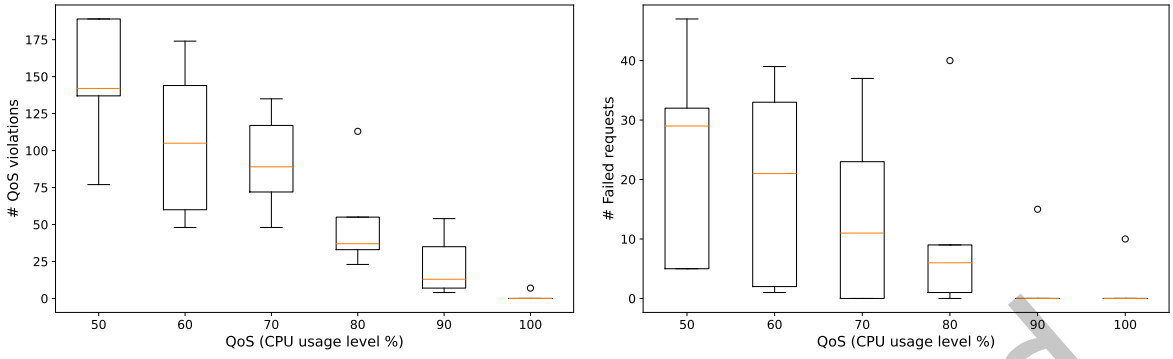

Fig. 11. RaspberryPi testbed for the use case evaluation with three Edge Clusters and an Experiments Controller node.

We generate the background workload using Locust, a workload generator tool [12]. We send multiple requests at an interval to all edge nodes with random images from the test dataset. Each node in a cluster receives incoming requests from $(1, 4)$ number of concurrent users, randomly chosen, and each user generates a new request within an interval of $(1, 15)$ seconds. This is important to evaluate our edge monitoring system in a dynamic workload environment.

Finally, the overall use case workflow is set up as follows. We extract 100 random user location entries from the user location dataset, and for each user location, the task is to offload computationally expensive tasks (e.g., object detection) to the nearest edge node. To do that, we need to know the suitable edge nodes that satisfy the user requirements, such as the availability and usage level of the nearest edge node, so that the user application can offload its computational task.

The DEMon framework is designed to provide a robust monitoring system for volatile edge environments. To accommodate high volatility due to node failures, our setup defines varying levels of failure: $failure\_rate = \{50, 60, 70, 80, 90\}$. Simulating a scenario where 90% of the total nodes fail represents an extremely volatile system. It is important to note that when nodes rejoin the system, the recovery time (i.e., gossip convergence) remains unchanged, namely $O(n \cdot m)$ (see Section 3.6), where $n$ represents the current number of active nodes in the system.

[12] https://locust.io/




(a) CPU QoS vs QoS violations.

(b) CPU QoS vs failed requests.

Fig. 12. Object detection offloading use case: Application performance with different level QoS settings.

Similarly, we select % CPU usage as the QoS parameter, as higher CPU usage would have higher computational latency for the offloading task and the lower the CPU threshold indicates the stricter QoS requirement (application task demands a node with low CPU usage). We set the CPU QoS to 50 – 100% with an interval of 10%. These configurations help us to evaluate the performance of DEMON under different failure rates.

Figure 12 shows the performance of offloading user requests to nearby edge nodes. We measure two important metrics: (1) *Number of QoS violations* - indicating how many times QoS is potentially violated, i.e., the nearest node has higher CPU usage than the expected QoS, forcing us to offload to the next nearest node. (2) *Number of failed requests* - indicating how many requests never find a suitable node in the edge network, thus, discarding the offloading requests. For both metrics, we present the average results across different failure rates configured as explained above. As seen in Figure 12a, when we have relaxed QoS (i.e., CPU threshold is 100%), the offloading is mostly done to the nearest node regardless of its usage level, without requiring our request to be redirected to the next nearest node. However, when we have strict QoS, a higher number of QoS violations are found. This means offloading is done to the next nearest suitable node in the edge environment. For instance, with a CPU threshold of 50%, the number of QoS violations had min=77, max=189, and mean=146.80. It is important to note that, to make such a decision, only one monitoring query is made to the nearest edge node initially. In fact, DEMon provides monitoring information of other nodes that are in the network. In other words, if no such monitoring system is present, we would have to make 147 extra queries to individual nodes in the worst case, while our system require one query.

Figure 12b shows the number of failed requests with different levels of QoS. The strict QoS setting (CPU threshold of 50%) results in an extremely high number of failed requests, with min=5, max=47, and mean=23.60. Similarly, when we set relaxed QoS (CPU threshold of 100%), the least number of failed requests are observed, with min=0, max=10, and mean=2. In summary, the results obtained by the use case implementation show how DEMon enables efficient retrieval of decentralized monitoring data, supporting critical applications in volatile edge environments.

## 5.5 Application in Real-world Scenarios

This section discusses the real-world applications of DEMon, a framework designed to deploy decentralized monitoring systems in highly volatile edge environments. DEMon is particularly beneficial in IoT-edge settings, where devices operate in highly mobile and unreliable conditions, such as poor network connectivity or unstable

power supplies. DEMon facilitates the monitoring of dynamic systems by eliminating a central point of failure. As a concrete use case, we demonstrate in Section 5.4 that decentralized monitoring information is a useful mobile edge environment for scheduling latency-sensitive ML tasks.

In broader application scenarios, decentralized monitoring systems like DEMon have applications in several resource and application management tasks in distributed systems. For instance, in runtime verification systems [17], where a verdict has to be made based on the global trace of monitored resource metrics; DEMon enables the development of such second-order applications based on decentralized monitoring information that not only disseminates information quickly but also allows for the retrieval of consistent (eventual) data in a decentralized manner. Moreover, systems like DEMon are also often needed in use cases like mobile ad-hoc networks and computing [39], software-defined networks [41], and robotic systems [31] for coordination and communication based on monitoring data.

### 5.6 Threats to Validity

We anticipate the following threats to our study. First, we have developed a prototype of our framework and conducted evaluations through emulation and on a small-scale testbed. However, this implementation is not directly transferable to real-world deployments due to additional critical components that must be considered. These include establishing secure communication (such as authentication and authorization) and creating static bootstrap nodes within the network, among others. Second, for the sake of rapid prototyping and ease of use, we chose Python and REST APIs for implementation and method evaluation. We recognize that further performance enhancements could be achieved by opting for more resource-efficient programming languages (e.g., C/C++ or Go) and communication frameworks (e.g., gRPC). Finally, to mitigate statistical bias in our results, we repeated each experiment three times and reported the mean value for each data point. Additionally, we have included standard deviations in the plots, where applicable, to provide a measure of variability.

Nevertheless, our methodological approach proves that DEMon is particularly effective in highly volatile edge environments where failures are common and reliable centralized monitoring is not feasible.

## 6 RELATED WORK

Table 3. A comparison of related works

| | System | Characteristics | Implementation/Evaluation Method | Communication Protocol | Storage Architecture | Trustworthiness |
|---|---|---|---|---|---|---|
| Wuhib et al. [48] | G-GAP | Monitoring of network-wide aggregates | Simulation using SIMPSON simulator and traces | Gossip | Centralized | ✗ |
| Ward et al. [47] | N/A | Monitoring IaaS cloud resources | Simulation | Layered Gossip | Partially centralized | ✗ |
| Taherizadeh et al. [40] | N/A | Network monitoring for real-time edge services | Implemented with JCatascopia framework | N:1 communication | Centralized | ✗ |
| Battula et al. [6] | SCB | Fog monitoring using support confidence-based technique | Implemented emulation tool in Java and use case evaluation | Hierarchical regions (clusters) with centralized server | Centralized | ✗ |
| Grogman et al. [22] | PyMon | A tool to collect container statistics running on heterogeneous nodes | Implementation using extending "monit" tool | N:1 communication | Centralized | ✗ |
| Brandon et al. [9] | FMone | Monitoring of edge resources based on use case-specific workflow | Prototype implementation and evaluation on Grid5K environment | Hierarchical regions (clusters) with centralized servers | Centralized | ✗ |
| Colombo et al. [12] | AdaptiveMon | Self-adaptive P2P monitoring of Fog | Implemented in C++ extending FogMon2 | P2P communication with leader and followers | Partially centralized | ✗ |
| Forti et al. [19, 21] | FogMon | Distributed Monitoring for Fog infrastructures | TRL5 implementation using C++ | P2P topology with leaders and followers | Partially centralized | ✗ |
| **Our Work** | **DEMon** | **A completely decentralized monitoring system for volatile edge** | **Prototype implementation with Python, evaluation in emulation and use case setup** | **Gossip** | **Decentralized** | ✓ |

Traditional monitoring systems in distributed computing are mainly centralized systems, where a set of dedicated servers periodically collect data from all the resources either through *push* or *pull* based APIs. Cloud monitoring tools such as Google's Borgmon [7], or Prometheus [42] provide flexibility to store data on local disk or remote storage; their control plane (e.g., alert management and query processing) is still centralized. Similar design principles are followed in High-Performance Computing (HPC) monitoring systems. However, existing Edge Computing platforms depend on a centralized monitoring system consequent to the design principle of

edge systems. For instance, Kubernetes [13] and its variants, such as KubeEdge, which are popularly used in edge infrastructures for resource and application management, use a centralized control plane and monitoring system. Such approaches are infeasible in critical edge infrastructures requiring strict latency and reliability under failures.

**Edge monitoring.** Researchers have made many efforts to address the challenges in Edge Computing monitoring. Grogman et al. [22] propose PyMon, a monitoring system targeting container-based computing architectures with a small resource footprint. The solution is primarily targeted at IoT-based single-board edge devices. The system was built as a set of container images providing monitoring service. Similarly, Taherizadeh et al [40] propose a network monitoring approach for data streaming applications, where each edge node hosts a monitoring probe and pushes the data to a centralized time-series database servers. It considers QoS metrics such as delay, packet loss, throughput, and jitters important for streaming applications. Brandon et al. [9] propose FMonE, designing a monitoring solution that considers elasticity and resiliency, addressing the unique challenges of edge systems. The prototype system uses Docker containers to build monitoring agents. However, storage and information processing depend on centralized database systems for specific application domains. All these solutions, including FMonE, either aggregate data from several edge sites with a central controller, or push monitored data to the Cloud.

Other works explored self-adaptive monitoring for multi-tier Fog computing systems. In particular, Forti et al. [19] propose FogMon, a hierarchical P2P monitoring system where lower-tier nodes are dedicated as Followers and higher-tier nodes as Leader nodes. The inherited self-adaptive P2P architecture characteristics make it suitable for volatile Fog-Edge environments. Similarly, Colombo and Tundo, et al. [12] propose AdaptiveMon, a self-adaptive P2P monitoring system that can dynamically change its behavior according to collected monitoring indicators by leveraging a rule-based expert system. Nevertheless, both these approaches still exhibit a partially centralized architecture in the form of Leader nodes, and they are only feasible when multi-tier deployments exist with Edge, Fog, and Cloud nodes.

Battula et al. [6] proposed a fog monitoring framework that utilizes a Support and Confidence-Based (SCB) technique to optimize resource usage of monitoring services. Their method takes into account the contextual information of resources, such as the current battery power of fog devices, to determine their participation in the monitoring service. The use case is assessed by grouping fog devices under a fog leader, with leaders sharing the monitoring data with a centralized server.

Gaglianese et al. [20, 21] extended the FogMon [19] framework, adding new features and providing a TRL5 level prototype system for monitoring fog devices. This extension follows FogMon's architecture for information dissemination and offers improved accuracy in monitoring data and fault resiliency with improved management of different types of failures. The proposed approach was evaluated on up to 40 edge nodes deployed on the Fed4Fire testbed. An extensive list of monitoring services in the fog environment and their comparative study is presented in [13]. These studies rely on partially centralized architectures that are suitable for the fog environment with reasonable reliability. We provide a brief comparison of the most relevant related work in Table 3. As observed, our work adopts a completely decentralized approach in the design of the monitoring service for the edge environment, ensuring trustworthy retrieval of monitoring data from distributed storage.

**Gossip protocol in monitoring.** Gossip protocols have wide applications in many domains. Especially various studies have used the Gossip protocol for monitoring networks and distributed systems. Ward et al. [47] propose a solution to monitor large-scale cloud systems with layered Gossip protocols. The authors group the resources into multiple layers, from a group of virtual machines to multi-region data centers. The multi-layered inter-group and inter-cloud communication protocol architecture are considered to match the bandwidth requirement, i.e., high-speed network connection within a data center, less reliable, and low-speed networks available between data center regions. The Gossip protocol is used to spread the information across all the virtual machines. However,

[13] https://kubernetes.io

how the information is stored and retrieved in the system is unclear. Similarly, Van et al. [44] propose Astrolabe, one of the earlier systems for distributed information management based on the Gossip protocol. It is mainly designed for resource monitoring, management, and data mining. With the combination of P2P Gossip protocol, mobile code, and SQL query language, the system is implemented to manage the data collection, storage, and aggregation in real-time. It organizes resources in hierarchical domains (e.g., Domain Name System. Furthermore, Wuhib et al. [48] use the Gossip protocol to monitor network-wide aggregate metrics (e.g., average, min, and max). Gossip-based protocols are used in other distributed systems tasks such as in failure detection [45] and network size and density estimation [14]. In these works, the main focus is on aggregating the information of resources and they do not address a wider spectrum of monitoring requirements, including information dissemination and retrieval methods. The decentralized databases such as Cassandra use the gossip protocol for implementing internode communication within data centers. However, these distributed databases still operate with a centralized control plane for communication coordination. Similarly, other decentralized databases, including weavedb[14] and orbited [15], are heavyweight since they use compute-heavy logics like blockchains and are mainly designed to store application business data. Thus, these Off-the-shelf databases are unsuitable for edge monitoring systems.

However, the primary objective of our work is to provide a framework and techniques for a self-adaptive, self-configurable, and trustworthy monitoring system applicable to volatile edge environments, without limiting to specific application tasks. The individual components, such as querying languages of related work, act as complementary solutions in our approach.

## 7 CONCLUSIONS AND FUTURE DIRECTIONS

Self-adaptive and decentralized monitoring systems are necessary for efficient management of the edge environments. However, existing approaches are centralized in architecture, which increases information storage and retrieval latency and creates failure bottlenecks, making them infeasible for highly-volatile edge environments.

In this work, we presented DEMon, a monitoring system designed to operate autonomously without any external controller and configuration manager, which stores monitoring data in a decentralized fashion. DEMon enables efficient information spreading by leveraging a Gossip-based protocol, and it provides a trustable retrieval mechanism for distributed information with a leaderless quorum consensus technique. Additionally, it allows hyper-parameter configuration based on the system properties. We implemented the proposed framework as a lightweight and interoperable container-based system, and we performed an extensive evaluation with both a simulated large-scale edge environment and an in-lab testbed. Our experimental results show that DEMon quickly spreads the monitored information, and can retrieve monitoring information even when most of the nodes fail.

In the future, we plan to extend the query capabilities of the system by integrating a query language for extracting granular data.



## REFERENCES

[1] Krzysztof R. Apt, Eryk Kopczynski, and Dominik Wojtczak. 2017. On the Computational Complexity of Gossip Protocols. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 765–771.

[2] Ahmed Arafa, Roy D Yates, and H Vincent Poor. 2019. Timely cloud computing: Preemption and waiting. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 528–535.

[3] Atakan Aral and Ivona Brandić. 2020. Learning spatiotemporal failure dependencies for resilient edge computing services. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1578–1590.

[4] Rasool Azimi and Hedieh Sajedi. 2018. A decentralized gossip based approach for data clustering in peer-to-peer networks. *J. Parallel and Distrib. Comput.* 119 (2018), 64–80.

[14] https://github.com/weavedb/weavedb

[15] https://github.com/orbitdb/orbitdb

[5] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. 2023. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 215–231.

[6] Sudheer Kumar Battula, Saurabh Garg, James Montgomery, and Byeong Kang. 2020. An Efficient Resource Monitoring Service for Fog Computing Environments. *IEEE Transactions on Services Computing* 13, 4 (2020), 709–722. https://doi.org/10.1109/TSC.2019.2962682

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.".

[8] Ken Birman. 2007. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review* 41, 5 (2007), 8–13.

[9] Álvaro Brandón, María S Pérez, Jesus Montes, and Alberto Sanchez. 2018. Fmone: A flexible monitoring solution at the edge. *Wireless Communications and Mobile Computing* 2018 (2018).

[10] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7. Portland, OR, 343477–343502.

[11] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS Netto, et al. 2018. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)* 51, 5 (2018), 1–38.

[12] Vera Colombo, Alessandro Tundo, Michele Ciavotta, and Leonardo Mariani. 2022. Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments. In *2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 156–166.

[13] Breno Costa, João Bachiega, Leonardo Rebouças Carvalho, Michel Rosa, and Aleteia Araujo. 2022. Monitoring fog computing: A review, taxonomy and open challenges. *Computer Networks* 215, C (oct 2022), 19 pages. https://doi.org/10.1016/j.comnet.2022.109189

[14] T. Darwish and K. Abu Bakar. 2015. Traffic density estimation in vehicular ad hoc networks: A review. *Ad Hoc Networks* 24 (2015), 337–351.

[15] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1–12.

[16] Hans Ditmarsch, Davide Grossi, Andreas Herzig, Wiebe van der Hoek, and Louwe B Kuijer. 2016. Parameters for epistemic gossip problems. In *LOFT 2016-12th Conference on Logic and the Foundations of Game and Decision Theory*.

[17] Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring decentralized specifications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 125–135.

[18] Stefano Forti, Gian-Luigi Ferrari, and Antonio Brogi. 2020. Secure Cloud-Edge Deployments, with Trust. *Future Generation Computer Systems* 102 (2020), 775–788.

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

[20] Marco Gaglianese, Stefano Forti, Federica Paganelli, and Antonio Brogi. 2022. Lightweight Self-adaptive Cloud-IoT Monitoring across Fed4FIRE+ Testbeds. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6. https://doi.org/10.1109/INFOCOMWKSHPS54753.2022.9798259

[21] Marco Gaglianese, Stefano Forti, Federica Paganelli, and Antonio Brogi. 2023. Assessing and enhancing a Cloud-IoT monitoring service over federated testbeds. *Future Generation Computer Systems* 147 (2023), 77–92.

[22] Marcel Großmann and Clemens Klug. 2017. Monitoring Container Services at the Network Edge. In *2017 29th International Teletraffic Congress (ITC 29)*, Vol. 1. 130–133.

[23] Xiangwang Hou, Zhiyuan Ren, Wenchi Cheng, Chen Chen, and Hailin Zhang. 2019. Fog Based Computation Offloading for Swarm of Drones. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 1–7.

[24] Shashikant Ilager, Jakob Fahringer, Samuel Carlos de Lima Dias, and Ivona Brandic. 2022. DEMon: Decentralized Monitoring for Highly Volatile Edge Environments. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. 145–150.

[25] Anne-Marie Kermarrec and Maarten Van Steen. 2007. Gossiping in distributed systems. *ACM SIGOPS operating systems review* 41, 5 (2007), 2–7.

[26] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17 (2015), 184–206. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian.

[27] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. 2018. Optimal edge user allocation in edge computing with variable sized vector bin packing. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*. Springer, 230–245.

[28] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4* (2001), 51–58.

[29] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. 2019. Computation Offloading Toward Edge Computing. *Proc. IEEE* 107, 8 (2019), 1584–1607.

[30] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-Time Object Detection for Mobile Augmented Reality. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 16 pages.

[31] Siddharth Mayya, Pietro Pierpaoli, and Magnus Egerstedt. 2019. Voluntary retreat for decentralized interference reduction in robot swarms. In *2019 international conference on robotics and automation (ICRA)*. IEEE, 9667–9673.
[32] Talal H Noor, Quan Z Sheng, Sherali Zeadally, and Jian Yu. 2013. Trust management of services in cloud environments: Obstacles and solutions. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–30.
[33] Rihards Olups. 2016. *Zabbix Network Monitoring*. Packt Publishing Ltd.
[34] Chanwon Park and Jemin Lee. 2020. Mobile edge computing-enabled heterogeneous networks. *IEEE Transactions on Wireless Communications* 20, 2 (2020), 1038–1051.
[35] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.
[36] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. 2019. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*.
[37] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*. 38–54.
[38] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.
[39] Dominik Stingl, Christian Gross, Leonhard Nobach, Ralf Steinmetz, and David Hausheer. 2013. BlockTree: Location-aware decentralized monitoring in mobile ad hoc networks. In *38th Annual IEEE Conference on Local Computer Networks*. 373–381. https://doi.org/10.1109/LCN.2013.6761269
[40] Salman Taherizadeh, Ian Taylor, Andrew Jones, Zhiming Zhao, and Vlado Stankovski. 2017. A Network Edge Monitoring Approach for Real-Time Data Streaming Applications. In *Economics of Grids, Clouds, Systems, and Services*, José Ángel Bañares, Konstantinos Tserpes, and Jörn Altmann (Eds.). Springer, Cham, 293–303.
[41] Gioacchino Tangari, Daphne Tuncer, Marinos Charalambides, and George Pavlou. 2017. Decentralized monitoring for large-scale Software-Defined Networks. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 289–297. https://doi.org/10.23919/INM.2017.7987291
[42] James Turnbull. 2018. *Monitoring with Prometheus*. Turnbull Press.
[43] Hans van Ditmarsch, Jan van Eijck, Pere Pardo, Rahim Ramezanian, and François Schwarzentruber. 2017. Epistemic protocols for dynamic gossip. *Journal of Applied Logic* 20 (2017), 1–31.
[44] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM transactions on computer systems (TOCS)* 21, 2 (2003), 164–206.
[45] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 1998. A gossip-style failure detection service. In *Middleware'98*. Springer.
[46] Tian Wang, Lei Qiu, Arun Kumar Sangaiah, Anfeng Liu, Md Zakirul Alam Bhuiyan, and Ying Ma. 2020. Edge-computing-based trustworthy data collection model in the internet of things. *IEEE Internet of Things Journal* 7, 5 (2020), 4218–4227.
[47] Jonathan Stuart Ward and Adam Barker. 2013. Monitoring large-scale cloud systems with layered gossip protocols. *arXiv preprint arXiv:1305.7403* (2013).
[48] Fetahi Wuhib, Mads Dam, Rolf Stadler, and Alexander Clemm. 2007. Robust Monitoring of Network-wide Aggregates through Gossiping. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. 226–235.