Nuno Morais

Bachelor in Computer Science and Engineering

DeMMon
Decentralized Management and Monitoring Framework

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science

Adviser: João Leitão
Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

⟨6⟩, ⟨2021⟩

DeMMon Decentralized Management and Monitoring Framework

Acknowledgements

The development of the work presented in this document would not have been possible without the help of some people and institutions that deserve the dearest acknowledgements.

First, I would like to thank the mentorship provided by my advisor, João Leitão, that, through example, inspired me to become a harder working person, and that through his insight and patience guided me to create this work.

Second, I would like to thank Bruno Anjos for the contributions provided to the developed benchmarking application, as well as for providing great company and coming into the lab with a smile throughout a rather unusual year due to COVID. Secondly, I would like to extend my thanks to Pedro Akos, for being a great colleague that was always ready to step in and help, providing great insights into my work as well as providing tools that helped benchmark the developed work.

Then, I would like to thank the Department of Informatics of the NOVA University of Lisbon and the NOVA LINCS research centre for providing a framework with mentorship and tools that greatly helped develop an ever-increasing interest in the area.

I also would like to extend a personal thank you to my partner, Marta Carlos, for making this thesis possible through the always enthusiastic support provided, given with a bright smile, even during the grimmest days of this, rougher than usual, year.

Then, I would like to thank my friends and family for always supporting me and encouraging me to follow what makes me happy.

Finally, the work presented in this thesis was partially supported by FCT through NOVA LINCS (grant UIDB/04516/2020) and NG-STORAGE (PTDC/CCIINF/32038/2017). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Abstract

The centralized model proposed by the Cloud computing paradigm mismatches the decentralized nature of mobile and IoT applications, given the fact that most of the data production and consumption is performed by end-user devices outside of the Data Center (DC). As the number of these devices grows, and given the need to transport data to and from DCs for computation, application providers incur additional infrastructure costs, and end-users incur delays when performing operations.

These reasons have led us into a post-cloud era, where a new computing paradigm arose: Edge Computing. Edge Computing takes into account the broad spectrum of devices residing outside of the DC, closer to the clients, as potential targets for computations, potentially reducing infrastructure costs, improving the quality of service (QoS) for end-users and allowing new interaction paradigms between users and applications.

Managing and monitoring the execution of these devices raises new challenges previously unaddressed by Cloud computing, given the scale of these systems and the devices' (potentially) unreliable data connections and heterogenous computational power. The study of the state-of-the-art has revealed that existing resource monitoring and management solutions require manual configuration and have centralized components, which we believe do not scale for larger-scale systems.

In this work, we address these limitations by presenting a novel Decentralized Management and Monitoring ("DeMMon") system, targeted for edge settings. DeMMon provides primitives to ease the development of tools that manage computational resources that support edge-enabled applications, decomposed in components, through decentralized actions, taking advantage of partial knowledge of the system. Our solution was evaluated to amount its benefits regarding information dissemination and monitoring capabilities across a set of realistic emulated scenarios of up to 750 nodes with variable failure rates. The results show the validity of our approach and that it can outperform state-of-the-art solutions regarding scalability and reliability.

Keywords: Edge Computing, Resource Management, Resource Monitoring, Topology Management, Middleware

Resumo

O modelo de computação centralizado utilizado no paradigma da Computação na Nuvem apresenta limitações no contexto de aplicações no domínio da Internet das Coisas e aplicações móveis. Neste, os dados são produzidos e consumidos maioritariamente por dispositivos que se encontram na periferia da rede. Visto isto, transportar estes dados de e para os centros de dados impõe carga excessiva nas infraestruturas de rede que ligam os dispositivos aos centros de dados, aumentando a latência de respostas e diminuindo a qualidade de serviço para os utilizadores.

Para combater estas limitações, surgiu o paradigma da Computação na Periferia, este paradigma propõe a execução de computações, e potencialmente armazenamento de dados, em dispositivos fora dos centros de dados, mais perto dos clientes, reduzindo custos e criando um novo leque de possibilidades para efetuar computações distribuídas mais próximas dos dispositivos que produzem e consomem os dados.

Contudo, gerir e supervisionar a execução desses dispositivos levanta obstáculos não equacionados pela Computação na Nuvem, como a escala destes sistemas, ou a variabilidade na conectividade e na capacidade de computação dos dispositivos que os compõem. O estudo da literatura revela que ferramentas populares para gerir e supervisionar aplicações e dispositivos possuem limitações para a sua escalabilidade, como por exemplo, pontos de falha centralizados, ou requerem a configuração manual de cada dispositivo.

Nesta dissertação, propõem-se uma nova solução de monitorização e disseminação de informação descentralizada. Esta solução oferece operações que permitem recolher informação sobre o estado do sistema, de modo a ser utilizada por soluções (também descentralizadas) que gerem aplicações especializadas para executar na periferia da rede. A nossa solução foi avaliada em redes emuladas de várias dimensões com um máximo de 750 nós, no contexto de disseminação e de monitorização de informação. Os nossos resultados mostram que o nosso sistema consegue ser mais robusto ao mesmo tempo que é mais escalável quando comparado com o estado da arte.

Palavras-chave: Computação na periferia, Computação distribuída, Gestão de recursos, Monitorização, Gestão de topologias de redes

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Nowadays, the Cloud Computing paradigm is the standard for the development, deployment, and management of services for most software systems present in our everyday life. Google Apps, Amazon, Twitter, among many others, are deployed on some form of cloud infrastructure and benefit from cloud-based services. Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and software systems in the data centers that provide those services [4]. This paradigm enables the illusion of unlimited computing power, which revolutionized the way companies and developers design, develop, maintain and manage their online applications, as well as the expectations that users have from them.

However, the centralized model proposed by the Cloud Computing paradigm mismatches the needs of many types of applications such as latency-sensitive applications, interactive mobile applications, and IoT applications [38]. All these application domains are characterized by having data being generated and accessed (predominantly) by end-user devices. When the computation resides in the data center (DC), far from the source of the data, challenges may arise: from the physical space needed to contain all the infrastructure, the increasing amount of bandwidth needed to support the information exchanges as well as the latency in communication from the clients to the DC. All of these challenges have directed us into a new computing paradigm: *Edge Computing*.

Edge computing addresses the increasing need for enriching the interaction between cloud computing systems and interactive/collaborative web and mobile applications [19] by taking into consideration computing and networking resources which exist beyond the boundaries of DCs, closer to the edge of systems [35] [56]. This paradigm also aims at enabling the creation of systems that could otherwise be unfeasible with Cloud Computing: Google's self-driving car generates 1 Gigabyte every second [55], and a Boeing 787 produces data at a rate close to 5 gigabytes per second [16], which would be impossible to transport and process in real-time (e.g., towards self-driving) if the computations were to be carried exclusively in a DC.

By taking into consideration all the devices which are external to the DC, as these range from Edge Data Centers to 5G towers and mobile devices, we are faced with a huge increase in the number and diversity of computational devices, that contrary to the cloud, have a wide range of computational capacity, and limited and (potentially) unreliable connections. Given this, we believe developing an efficient resource management/sharing platform that uses these devices toward generic computation is an open challenge to fully accomplish in Edge Computing.

1.2 Motivation

Resource management/sharing platforms are extensibly used in Cloud systems (e.g. Mesos [23], Yarn [63], Omega [54], among others), whose high-level functionality consists of: (1) federating all the devices and tracking their state and utilization of computational and networking resources; (2) keeping track of resource demands which arise from different tenants; (3) performing resource allocations to satisfy the needs of such tenants; (4) adapting to dynamic workloads such that the system remains balanced and system policies as well as performance criteria can be ensured.

Most popular resource management and sharing platforms are tailored towards small numbers of homogenous resource-heavy devices, which rely on a centralized system component that performs resource allocations with a global knowledge of the system (including available computational resources, their usage, workloads received per each hosted application, e.t.c). Although this system architecture heavily simplifies the management of the resources, we argue that such systems, as they are often plagued by a central point of failure and a single point of contention, have hindered scalability and fault-tolerance, making them unsuitable for the more heterogeneous and larger infrastructure that can be leveraged by Edge Computing systems.

Instead, to achieve general-purpose computation in Edge systems, we argue in favour of decentralized management/sharing systems, composed of multiple components, organized in a flexible hierarchical way, that perform resource management decisions supported by partial and localized knowledge of the system state. Because building such a platform from scratch is not trivial, and as we believe that in such a system, the accuracy and freshness of the information available to each component (which includes but is not exclusive to the execution of components or services), dictates how efficiently they manage resources, we focus on that particular task: **decentralized data collection and aggregation**.

Hence, the goal of this work is to propose a novel solution that provides efficient decentralized data collection and aggregation primitives over multiple nodes located in and outside of the DC. It is our end goal to ease the creation of a new generation of fully decentralized resource management solutions that employ partial and localized knowledge, paving the way to more decentralized and effective solutions to manage complex edge infrastructures, enabling to improve the performance of future edge-enabled

applications.

1.3 Contributions

The contributions which arose from the conducted work are as follows:

1. A smaller contribution that derived from the work conducted in the context of this thesis which consists in a port to Golang of Babel [1], a framework for building distributed systems, used to build our solution. This contribution has some additions focused on latency measurement and fault detection.

2. A distributed monitoring framework, built for decentralized resource management systems, composed of three main components:

   a) A novel overlay protocol which strives to build a logical multi-tree-shaped logical network using both bandwidth and node latency as heuristics for defining the topology of the network. This protocol is fully decentralized and fault-tolerant, with its configuration being only a set of static nodes: the roots of the trees.

   b) A distributed aggregation protocol, which uses the connections made available by the overlay protocol's tree structure to perform efficient on-demand and decentralized information collection.

   c) An API that allows resource management applications to insert, process, and retrieve time-series data to and from a DeMMon node. This API also offers operations that allow the collection and processing of information from other DeMMon nodes in the network.

   d) An experimental evaluation of the membership protocol against popular alternatives found in the state of the art, where their fault tolerance, the ability to improve the network cost, and their capacity to perform information dissemination reliably is studied.

   e) An experimental evaluation of the monitoring protocol against different Prometheus [47] configurations. This evaluation focuses on the accuracy of the collected monitoring values over time, as well as the cost for networking/processing the information.

3. A proposed benchmark in the form of an edge-enabled application composed by multiple loosely coupled micro-services, tailored to evaluate the performance of resource management platforms. In this benchmark, geographical proximity leads to a significant improvement of QOS for the end-user, favouring resource management platforms that optimize placement of their services closer to the client (i.e. applications that take advantage of edge computing).

1.4 Document structure

The remaining of this document is structured as follows:

Chapter 2 studies related work that is relevant to the overall goal of the work presented in this thesis: we begin by analyzing similar paradigms to Edge Computing, the devices which compose these environments, and execution environments for edge-enabled applications. We also discuss strategies towards federating various devices in an abstraction layer and study search strategies to find resources in this layer. Finally, we cover monitoring and management of system resources.

Chapter 3 covers the design and implementation of the initial contribution, that as previously mentioned, consists of a port to Golang of Babel [1], which in turn was used to build our solution.

Following, chapter 4 explains the implementation of the developed solution, beginning with the overlay protocol, followed by the aggregation protocol that uses the overlay protocols' connections, and lastly, we cover the monitoring module of our system that stores and serves information as time-series.

After, in chapter 5, we cover the design and implementation of the previously mentioned edge-enabled benchmarking application targeted for testing decentralized resource management platforms.

In chapter 6, we provide the results of our experimental evaluation regarding the devised overlay protocol and aggregation protocol.

Finally, in chapter 7, we draw conclusions from the conducted work and discuss the future work we intend to pursue to further improve our solution.

2

Related Work

The goal of this chapter is to present the work in the state of the art that is related to our objectives. We begin by presenting what we believe to be four main high-level requirements of a decentralized resource sharing platform, as denoted in Figure 2.1:

1. *Topology Management* consists in the study of how to organize multiple devices in a logical network such that they can cooperatively solve tasks. Efficiently managing the topology is an essential building block for achieving efficient operation of the remaining components.

2. *Resource Location and Discovery* focuses on how to efficiently index and locate resources in the aforementioned logical network. For example, in the context of resource sharing, resource discovery is paramount towards locating nearby devices which have enough (free) computing and networking capabilities to perform a certain task, or host a certain applicational component or service.

3. *Monitoring and aggregation* studies how to track device metrics and how to efficiently compress the size of those metrics through (possibly decentralized) aggregation. Then, the compressed result may be used to improve the operation of a certain service or component.

4. *Resource Management* addresses how to efficiently manage system resources and schedule jobs (including hosting services, running a single task, among other types of jobs) across existing resources such that: (1) the system remains load-balanced; (2) jobs can operate efficiently; (3) jobs have (preferably) data locality; (4) resources are not wasted. While the work conducted in this thesis is tailored toward supporting this goal, this thesis does not aim at devising a complete scheduling solution, as that is a complete research line on its own. However, for completeness, in this chapter, we also discuss this aspect.

Considering the identified high-level requirements of such a system, and considering the goal of creating such a system for the edge environment, in the following sections we begin by covering the taxonomy of devices that compose the edge environment and

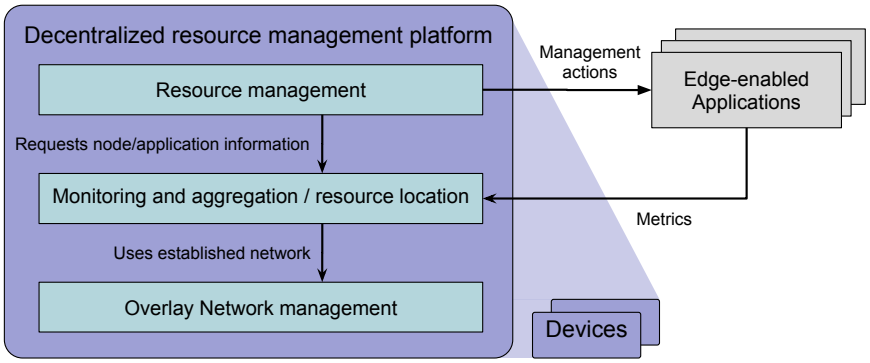

Figure 2.1: High-level components for a resource sharing platform

discuss how they can be employed towards the design of the proposed solution (Section 2.1).

Following, we study how to federate devices in an efficient abstraction layer that establishes an efficient topology (Section 2.2), we address how peers can efficiently organize to index and search for the resources they need (e.g. services, peers, computing power, among others), which in turn enables the delegation of particular application components (Section 2.3).

Next, in Section 2.4, we cover the current state-of-the-art regarding the collection and aggregation of metrics, which are paramount to summarize and enrich nodes with the necessary information to perform efficient resource management decisions in a decentralized manner. We study current aggregation practices and discuss relevant resource monitoring systems in the literature. For each system, we address its limitations and advantages for the edge environment. Lastly, as we believe it is necessary to gather a perception of the needs of current state-of-the-art resource management solutions, in Section 2.5 we briefly survey the related work in this area.

2.1 Edge Environment

In this section, we study the taxonomy of the devices which materialize edge environments and analyze, according to the literature, which computations each device can perform.

2.1.1 Edge Environment Taxonomy

According to Leitão et al. [35], edge devices may be classified according to three main attributes: **capacity** refers to computational, storage and connectivity capabilities of the device, **availability** consists in the probability of a device being reachable, and finally, **domain** characterizes the way in which a device may be employed towards applications, either by performing actions on behalf of users (user domain) or performing actions on

behalf of applications (applicational domain). Given that the concern of our work is towards building the underlying infrastructure for these applications, we will only focus on capacity and availability when classifying the taxonomy of the environment.

Table 2.1: Taxonomy of the edge environment

Level	Category	Availability	Capacity
L0	Cloud Data Centers	High	High
L1	ISP, Edge & Private DCs	High	High
L2	5G Towers	High	Medium
L3	Networking devices	High	Low
L4	Priv. Servers & Desktops	Medium	Medium
L5	Laptops	Low	Medium
L6	Mobile devices	Low	Low
L7	Actuators & Sensors	Varied	Low

Table 2.1 shows the proposed categories of edge devices, we assign levels to categories as a function of their distance from the cloud infrastructure.

Levels 0 and 1, composed of *cloud and edge DCs*, offer pools of computational and storage resources which can dynamically scale. Both of these options have high availability and large amounts of storage and computational power, as such, there is no limitations on the kinds of computations these devices can perform.

Levels 2 and 3 are composed of *networking devices*, namely *5G cell towers*, *routers*, *switches*, and *access points*. Devices in both levels have high availability, and can easily improve the management of the network, for example, by manipulating data flows among different components of applications (executing in different devices).

Levels 4 and 5 consist of *private servers*, *desktops* and *laptops*, devices in these levels level have medium capacity and medium to low availability. They can perform a varied amount of tasks on behalf of devices in higher levels (e.g. compute on behalf of smartphones, act as logical gateways or just cache data).

Levels 6 consists of *tablets* and *mobile devices*, which have low capacity, availability, and short battery life. Given this, they are limited in how they can perform contribute towards edge applications. Aside from caching user data, they may filter or aggregate of data generated from devices in level 7.

Finally, **level 7** consists of *actuators*, *sensors* and *things*, these devices are the most limited in their capacity, and enable limited forms of computation in the form of aggregation and filtering.

2.1.2 Discussion

Coincidently, the levels are correlated to the number of devices and their computational power, where higher levels tend to have more devices that are closer to the origin of the data while having lower computational power. Consequently, the higher the level, the harder it is to employ edge devices to support the execution of edge-enabled applications.

We believe the low availability and potential mobility of devices in higher levels make them unsuitable, as they could potentially be a source of instability in the system. Consequently, we believe only devices in levels 0-5 are potential candidates to integrate the system we intend to build, provided the remaining devices tend too low of an availability to be possible targets to, for example, host a service reliably. It is, however, important

to mention that employing devices in other levels as gateways for those devices can help circumvent this limitation. Hence, starting to establish a natural hierarchy on the way different application components interact and how information (calls or events) flow in such complex systems.

2.2 Topology Management

Provided with an overview of the taxonomy of the devices materializing edge environments, we now study the related work towards federating all these devices (that we also refer to as peers following the peer-to-peer (P2P) literature nomenclature) in an abstraction layer (an overlay network) that allows intercommunication, cooperation, and efficient resource discovery [62]. This Section provides context regarding the taxonomy of overlay networks, followed by a discussion of popular overlay network protocols and what we believe to be their strengths and limitations.

In a P2P system, peers contribute to the system with a portion of their resources to accomplish tasks otherwise unfeasible by an individual peer. Typically, this is achieved in a decentralized way, which means peers must establish neighbouring connections among themselves to enable information exchange which, in turn, enables progress towards the system goals.

Participants in a P2P system may know all other peers in the system, which is typically referred to as **full membership** knowledge, which is a popular approach in Cloud systems. However, as the system scales to larger numbers of peers, concurrently entering and leaving the system (a phenomenon called churn [58]), this information becomes costly to maintain up-to-date.

In order to circumvent the aforementioned challenges, a common alternative is to have peers only maintain a view of a subset of all peers in the system, which is called **partial membership**. This information is maintained by some membership algorithm that restricts neighbouring relations among peers. Partial membership solutions are attractive because they offer similar functionality to full membership systems while achieving more scalability and resiliency to churn. The closure of these neighbouring relations is what materializes an **overlay network**, a logical (i.e. at the applicational level) network that is defined above another network (e.g. the IP network).

2.2.1 Taxonomy of Overlay Networks

Overlay networks are logical networks that operate at the applicational level. These rely on an existing network (commonly referred to as the *underlay*) to establish neighbouring relations, where each participant typically only communicates directly with its overlay neighbours [62]. Overlays are commonly designed towards specific applicational needs. As such, their neighbouring relations may or may not follow some established logic. As

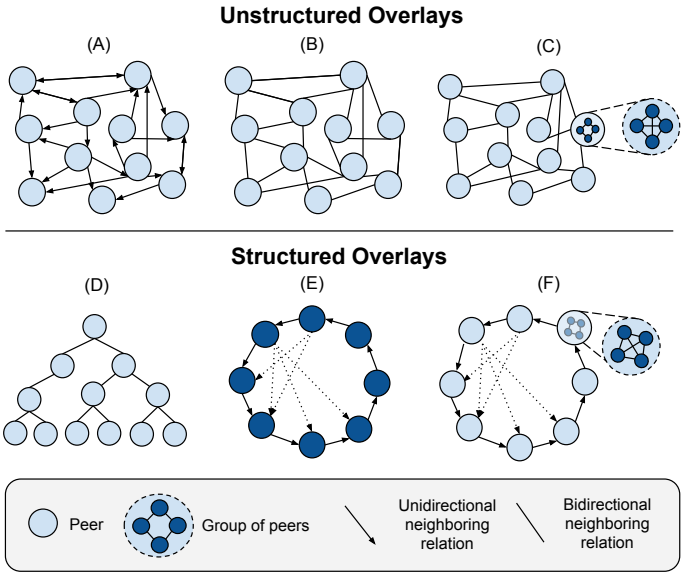

Figure 2.2: Examples Overlay Networks

illustrated in Figure 2.2, there are two main categories of overlays: **structured** and **unstructured**:

Unstructured Overlays

Unstructured overlays usually impose little to no rules in neighbouring relations. Nodes may pick random peers to be their neighbours or employ strategies to rank neighbours and selectively pick the best given particular criteria typically entwined with the needs of applications. A key factor of unstructured overlays is their low maintenance cost, given that nodes can easily create neighbouring relations, which eases the process of replacing failed ones. Consequently, this is the type of overlay which offers better resilience to churn.

In Figure 2.2, we illustrate three examples of unstructured overlay networks: (A) is a representation of an overlay network where the connections are unidirectional (e.g. Cyclon [30]), in this type of overlay, peers have no control over the status of incoming connections. Consequently, a peer may become isolated from the network without realizing it, which is undesirable.

Overlay (B) is similar to (A), however, its neighbouring connections are bidirectional. This means that a peer with a given number of outgoing connections must also have the correspondent number of incoming connections, diminishing the risk of the peer becoming disconnected from the overlay (this is the approach taken by HyParView [32] to achieve high reliability and fault-tolerance).

Lastly, (C) is a representation of an unstructured overlay where peers establish groups among themselves (such as Overnesia [37]). Grouping multiple devices into a group can provide benefits such as (1) failures can be quickly identified and resolved by other

members of the group; (2) nodes can replicate data within the group, leading to increased availability of that data; (3) groups can abstract groups of resources and internally manage their usage by, for example, offloading computational tasks within the group in a localized way.

Structured Overlays

Structured overlays enforce stronger rules towards neighbour selection (generally based on the identifiers of peers). As a result, the overlay generally converges to a certain topology known a *priori* (e.g., a ring, tree, hypercube, among others).

Figure 2.2 also illustrates three kinds of structured overlay networks: (D) corresponds to a tree, which are widely used to perform broadcasts (e.g., PlumTree [33]) because of the smaller message complexity required to deliver a message to all nodes, or to monitor the system state (if nodes in lower levels of the tree periodically send monitoring information [34] to upper levels in the tree, in turn, the root of the node has a global view of the collected monitoring information (e.g., Astrolabe [48])). However, trees are very fragile in the presence of faults [33].

Overlay (E) corresponds to an overlay typically expected to define Distributed Hash Tables (DHTs). These are extremely popular due to their effective applicational-level routing capabilities. In a DHT, peers employ a global coordination mechanism that restricts their neighbouring relations such that they can find any peer *responsible* for any given key in a limited number of steps (typically the logarithm of the system size). In this example (Figure E), the topology consists of a ring (which is the strategy employed by Chord [57]). It is important to mention that not all distributed hash tables rely on rings to perform effective routing. For example, in Kademlia [42], nodes organize as leaves across a binary tree.

Finally, the overlay denoted in (C) is similar to overlay (E), however each position of the DHT is made up of a virtual node composed of multiple physical nodes (which is the strategy employed by Rollerchain [46]). Because of this, routing procedures are still limited acording to the logarithm of system size, and also have the potential to be load-balanced. Furthermore, as the failure of a physical node does necessarily mean the failure of a virtual node, churn effects are mitigated.

2.2.2 Overlay Network Quality Metrics

If we look at an overlay network where connections between nodes represent edges and nodes represent vertices in a graph, we obtain a graph from which we may extract direct metrics to estimate overlay performance [62], we now enumerate some which we believe to be the most relevant to our goal:

1. **Connectivity**. This property is usually measured as a percentage, corresponding to the largest portion of the system that is connected, intuitively, a connected graph is

one where there is at least one path from each node to all other nodes in the system.

2. **Degree Distribution**. The degree of a node consists in the number of arcs that are connected to it. In a directed graph, there is a distinction between **in-degree** and **out-degree** of a node, nodes with a high in-degree value have higher reachability, while nodes with 0 in-degree cannot be reached. The out-degree of a node represents a measure of the contribution of that node towards the maintenance of the overlay topology.

3. **Average Shortest Path**. A path is composed by the edges of the graph that a message would have to cross to get from one node to other. The average shortest path consists in the average of all shorter paths between every pair of peers, to promote efficient communication patterns, is desirable that this value is as low as possible.

4. **Clustering Coefficient**. The clustering coefficient provides a measure of the density of neighbouring relations across the neighbours of links between a given node. It consists in the number of a node's neighbours divided by the maximum number of links that could exist between those neighbours. A high value of clustering coefficient means that there is a higher amount of redundant communication among nodes.

5. **Overlay Cost**. If we assume that a link in the overlay has a *cost*, (e.g. derived from latency), then the overlay cost is the sum of all the costs of the links that form the overlay.

2.2.3 Relevant examples of Overlay Networks

T-MAN [28] is a protocol to manage the topology of overlay networks, it is based on a gossiping scheme, and proposes to build a wide range of structured overlay networks (e.g., ring, mesh, tree, among others). To achieve this, T-MAN expects a cost function as an input to the protocol, then employed as a ranking method, applied iteratively by every node to compare the preference among possible neighbours.

Nodes periodically exchange their neighbouring sets with peers in the system and keep the nodes which rank higher according to the ranking method. A limitation of T-Man is that it does not ensure the stability of the in-degree of nodes during the optimization of the overlay, and consequently, the overlay may not remain connected.

Management Overlay Network [40] (MON) is an overlay network system aimed at facilitating the management of large distributed applications. This protocol builds on-demand overlay structures that allow users to execute instant management commands, such as query the current status of the application or push software updates to all the nodes. MON performs these procedures in an on-demand fashion such that it achieves a low maintenance cost when no commands are running.

This solution allows the on-demand construction of two types of Overlay Networks: trees and direct acyclic graphs. These overlays, in turn, can be employed towards aggregating monitoring data related to the status of the devices. Limitations from using MON are that the resulting overlays are susceptible to topology mismatch, and do not ensure connectivity. Furthermore, since the topologies are supposed to be short-lived, MON does not provide mechanisms for dealing with faults.

Hyparview [32] (Hybrid Partial View) gets its name from maintaining two exclusive views: the *active* and *passive* view, which are distinguished by their size and maintenance strategy.

The *passive view* is a larger view which consists of a random set of peers in the system, maintained by a periodic gossip protocol, where each peer sends a message to another random peer in their active view. This message contains a subset of the neighbours of the sending node and a time-to-live (TTL). The message is then forwarded randomly throughout the system until the TTL expires, updating the views of nodes it passes. In contrast, the *active view* consists of a smaller view (around log(n) in size), created during the bootstrap of the protocol. Each peer in the view has an associated TCP connection, which is then used as a bidirectional connection medium and failure detector. Whenever a node from the active view is detected as failed, it is replaced with one in the passive view.

Hyparview is often used as a *peer sampling service* for other protocols which rely on the connections from the active view to collaborate (e.g. PlumTree [33]). It achieves high reliability even in the face of a high percentage of node failures. However, its resulting topology is flat, which we believe to not be ideal for the taxonomy of edge environments we are considering. Furthermore, it may suffer from topology mismatch: given the random nature of neighbouring connections, the resulting neighbouring connections may be very distant in the underlying network.

X-BOT [36] is a protocol that constructs an unstructured overlay network where neighbouring relations are biased considering one metric. This metric is provided by an *oracle*, which is a component that exports a function, which accepts a pair of peers and attributes a cost to that neighbouring connection. This cost may take into consideration factors such as latency, ISP distribution, network stretch, among others.

The rationale X-BOT is as follows: nodes maintain active and passive views similar to Hyparview [32]. Then, nodes periodically trigger optimization rounds where they attempt to bias a portion of their connections according to the functions provided by the oracle. Although this protocol potentially addresses the previous concerns about the overlay topology mismatching the underlying network, it still proposes a flat topology, which, as previously mentioned, we believe is not adequate for the edge environment taxonomy.

Overnesia [37] is a protocol that establishes an overlay composed of fully connected groups of nodes, where all nodes within a group share the same identifier. Nodes join the system by sending a request to a bootstrap node, triggering a random walk. The

requesting node joins the group where its random walk terminates (either because it finds an underpopulated group or because the TTL expires).

Nodes enforce intra-group membership consistency through an anti-entropy mechanism where nodes within a group periodically exchange messages containing their view of the group. When a group detects that its size has become too large, it triggers a dividing procedure that splits the group in two halves. Conversely, when the group size has fallen below a certain threshold, nodes trigger a collapsing procedure. During it, each node takes the initiative to relocate itself to another group, resulting in the graceful collapse of the group. Finally, inter-group links are acquired by propagating random walks throughout the overlay.

As previously mentioned, establishing groups of nodes enable load-balancing, efficient dissemination of queries, and fault tolerance. Furthermore, it allows the abstraction of a set of physical resources into a single, unified, logical resource.

However, in systems with heterogeneous composition, system scalability limitations may arise as devices may have trouble maintaining the group view up-to-date and the active connections to all group members. Finally, the overlay may suffer from topology mismatch, as two nodes within the same group may be distant within the underlay.

Chord [57] is a well known structured overlay network where the protocol builds and manages a ring topology, similar to overlay (E) in Figure 2.2. Each node is assigned an m-bit identifier, uniformly distributed across the id space, and takes steps to fill its *finger table*. The finger table contains at most m entries, each ith entry of this table corresponds to the first peer that succeeds a certain peer n by 2^{ith} in the ring. This means that whenever the finger table is up-to-date, and the system is stable, lookups for any data piece only take logarithmic time to finish.

Although Chord provides a good trade-off between bandwidth and lookup latency, it has its limitations: peers do not learn routing information from incoming requests, links have no correlation to latency or traffic locality, and the overlay is highly susceptible to churn [39]. Finally, the ring topology is flat, which means that lower capacity nodes in the ring may become a limitation instead of an asset in the context of routing procedures.

Pastry [51] is another well known DHT which assigns a 128-bit node identifier (nodeId) to each peer in the system. The nodeIds are randomly generated, and consequently, are uniformly distributed in the 128-bit nodeId space. Routing procedures are forwarded to nodes whose nodeId shares a prefix that is at least one bit closer to the key, if there are no nodes available, nodes route messages towards the numerically closest nodeId. This routing procedure takes O(log N) routing steps, where N is the number of Pastry nodes in the system.

This protocol has been widely used as a building block for Pub-Sub applications such as Scribe [52] and file storage systems like PAST [14]. However, limitations from using Pastry arise from the use of a numeric distance function towards the end of routing procedures, which creates discontinuities at some nodeId values and complicates attempts at formal analysis of worst-case behaviour, in addition to establishing a flat topology that

mismatches the edge device taxonomy.

Tapestry [67] Is a DHT with similar behaviour to Pastry [51]. In this system, however, nodeIDs are represented using base b, where b is a parameter specified during configuration. In routing procedures, messages are incrementally forwarded to the destination digit by digit (e.g. ***8 -> **98 -> *598 -> 4598). Consequently, in stable conditions, routing procedures theoretically take $\log_b n$ hops to reach their destination, where b is the base of the ID space. Because nodes assume that the preceding digits all match the current node's suffix, nodes in Tapestry only need to keep a constant size of $\log N$ entries at each route level, consequently, nodes contain entries for a fixed-sized neighbour map of size b.

Kademlia [42] is a DHT where nodes are considered leaves distributed across a binary tree. Peers route queries and locate data pieces by employing an XOR-based distance function which is symmetric and unidirectional. Each node in Kademlia is a router where its routing tables consist of shortcuts to peers whose XOR distance is between 2^i by 2^{i+1} in the ID space, given the use of the XOR metric, "closer" nodes are those that share a longer common prefix.

The main benefits that Kademlia draws from this approach are: nodes learn routing information from receiving messages, there is a single routing algorithm for the whole routing process (unlike Pastry [51]) which eases formal analysis of worst-case behavior. Finally, Kademlia exploits the fact that node failures are inversely related to uptime by prioritizing nodes that are already present in the routing table.

Kelips [21] is a group-based DHT which exploits increased memory usage and constant background communication to achieve reduced lookup time and message complexity. Kelips nodes are split in k affinity groups split in the intervals $[0,k-1]$ of the identifier space, thus, with n nodes in the system, each affinity group contains $\frac{n}{k}$ peers. Within a group, nodes store a partial set of nodes contained in the same affinity group and a small set of nodes lying in foreign affinity groups. With this architecture, Kelips achieves O(1) time and message complexity in lookups, however, it has limited scalability when compared to previous DHTs, given the increased memory consumption ($O(\sqrt{n})$).

Rollerchain [46] is a protocol which establishes a group-based DHT by leveraging on techniques from both structured and unstructured overlays (Chord and Overnesia). In short, the Overnesia protocol materializes an unstructured overlay composed by logical groups of physical peers who share the same identifier. Then, the peer with the lowest identifier within each logical group joins a Chord overlay, obtains the adresses of other virtual peers, and distributes them among group members.

Rollerchain has the potential to enable a type of replication which has higher robustness to churn events when compared to other other replication strategies, however, there are limitations to this approach: (1) the load is unbalanced within members of each group, as only one node is in charge of populating and balancing the inter-group links; (2) similar to Chord, nodes do not learn from incoming queries, which contrasts with other DHTs such as Pastry; (3) the protocol has a higher implementation complexity and maintenance

cost when compared to a regular DHT.

2.2.4 Discussion

Unstructured overlays are an attractive option for federating large amounts of devices in heavily dynamic environments. They provide a low clustering coefficient, are flexible, and maintain good connectivity even in the face of churn. However, given their unstructured nature, they are limited in certain scenarios, for example, when trying to find a specific peer or resource in the system.

Conversely, distributed hash tables enable efficient routing procedures with very low message overhead, which makes them suitable for application-level routing. However, given their strict neighbouring rules, participating nodes cannot replace neighbours easily, which hinders the fault-tolerance of these types of topologies, in addition, given the fact that devices in edge environments have varied computational power and connectivity, they may become a limitation instead of an asset in the context of routing procedures.

2.3 Resource Location and Discovery

Resource location systems are one of the most common applications of the P2P paradigm [62], in a resource location system, a participant provided with a resource descriptor is able to query other peers and obtain an answer to the location (or absence) of that resource in the system within a reasonable amount of time. To do so, resource location systems employ search strategies, which depend on : (1) the structure the an overlay network (structured or unstructured). (2) on the characteristics of the resources to search (e.g. if there are many copies of it or not), and (3) on the desired results (e.g. if a single copy of a resource satisfies the query, or multiple are required).

In the context of resource management, if a peer wishes to offload computations to other peers, it must employ an efficient search strategy to find nearby available resources (e.g., storage capacity, computing power, among others) in order to offload computations. In this section, we cover resource location and discovery, starting with the taxonomy of querying techniques for P2P systems, followed by the study of how resources can be stored or indexed and looked up throughout the topologies studied in the previous section.

2.3.1 Querying techniques

Querying techniques consist of how peers describe the resources they need, these, according to [62], may be classified as: **(1) Exact Match queries**, these specify the resource to search by the value of a unique attribute (i.e., an identifier, commonly the hash of the value of the resource); the second querying methodology type is **(2) keyword queries**, that employ one or more keywords (or tags) combined with logical operators to describe

resources (e.g. "pop", "rock", "pop and rock"...); next, **(3) range queries** retrieve all resources whose value (or the value of a particular property)is contained within a given interval (e.g. "movies with 100 to 300 minutes of duration"); finally, **(4) arbitrary queries** aim to find a set of nodes or resources that satisfy one or more arbitrary conditions (e.g. looking for a set of resources encoded in a certain format).

Provided with a way of describing their resource needs, peers need strategies to index and retrieve the resources in the system, there are three popular techniques: **centralized**, **distributed over an unstructured overlay**, or **distributed over a structured overlay**.

2.3.2 Centralized Resource Location

Centralized resource location relies on one (or a group of) centralized peers that index all existing resources. This type of architecture greatly reduces the complexity of systems, as peers only need to contact a subset of nodes to locate resources.

It is important to notice that in a centralized architecture, while the indexation of resources is centralized, the resource access may still be distributed (e.g. a centralized server provides the addresses of peers who have the files, and files are obtained in a pure P2P fashion), a system which employs this architecture with success is BitTorrent [10].

Although centralized architectures are widely used nowadays, they lack the necessary scalability to index the large number of dynamic resources we intend to manage, and have limited fault tolerance to failures, making them unsuited for edge environments.

2.3.3 Resource Location on Unstructured Overlays

When employing an unstructured overlay for resource location, the resources are scattered throughout all peers in the system, consequently, peers need to employ distributed search strategies to find the intended resources. This is accomplished through disseminating messages containing these queries throughout the overlay. The dissemination of these messages can follow multiple strategies, we now cover there two popular approaches: **flooding** and **random walks** [62].

Flooding consists of peers eagerly forwarding queries to others in the system as soon as they receive them for the first time, the objective of flooding is to contact multiple distinct peers that may have the queried resource. One approach is **complete flooding**, which consists in contacting every node in the system, this guarantees that if the resource exists, it will be found. However, complete flooding is not scalable and incurs significant message redundancy. **Flooding with limited horizon** minimizes the message overhead by attaching a time to live (TTL) to messages that limit the number of times that messages can be retransmitted. However, there is a trade-off for efficiency: flooding with limited horizon does not guarantee that all resources will be found.

Random Walks are a dissemination strategy that attempts to minimize the communication overhead that is associated with flooding. A random walk consists of a message

with a TTL that is randomly forwarded one peer at a time throughout the network. Random walks may also attempt to bias their path towards peers that are more likely to have answers to the query [12], this technique is commonly reffered to in the literature as a **random guided walk**. A common approach to bias random walks is to use bloom filters [60], which are space-efficient probabilistic data structures that allow the creation of imprecise distributed indexes for resources.

First generation of decentralized resource location systems relied on unstructured overlays (such as Gnutella [20]) and employed simple broadcasts with limited horizon to query other peers in the system. However, as the size of the system grew, simple flooding techniques lacked the required scalability for satisfying the rising number of queries, which triggered the emergence of new techniques to reduce the number of messages per query, called **super-peers**.

Super-peers are peers which are assigned special roles in the system (often chosen in function of their capacity or stability). In the case of resource location systems, super-peers disseminate queries throughout the system. This technique is at the core of solutions such as Gia [9], employed towards effectively reducing the number of peers that have to disseminate queries on the second version of Gnutella [20].

SOSP-Net [17] (Self-Organizing Super-Peer Network) proposes a resource location system composed by regular peers and super-peers that effectively employs feedback concerning previous queries to improve the overlay network. Weak peers maintain links to super-peers which are biased based on the success of previous queries, and super-peers bias the routing of queries by taking into account the semantic content of each query.

However, even with super-peers, one problem that still remains in these systems is finding very rare resources, which requires flooding the entire overlay. To circumvent this, the third generation of resource location systems rely on Distributed Hash Tables to ensure that even rare resources in the system can be found within a limited number of communication steps.

2.3.4 Resource Location on Distributed Hash Tables

Resource location on structured overlays is often done by relying on the applicational routing capabilities of distributed Hash Tables (DHTs). In a DHT, peers use hash functions to generate node identifiers (IDS) often uniformly distributed over the ID space. Then, by employing the same hash function to generate resource IDs, and assigning a portion of the ID space to each node, peers are able to map resources to the responsible peers in a bounded number of steps, which makes them very suitable for (**exact match queries**) [62].

There are two popular techniques for storing resources in a DHT, the first approach is to store the resources locally and publish the location of the resource in the DHT. This way, the node responsible for the resource's key only stores the locations of other nodes in the system and the resource may be replicated among distinct nodes composing the

system. The second technique consists of transferring the resource to the responsible node in the DHT, although fewer nodes must keep the same value. It is, however, important to mention that this way the resources are not replicated, provided that with consistent hashing, all nodes with the same resource will publish the resource in the same location of the DHT.

2.3.5 Discussion

As mentioned previously, we believe centralized resource location systems are unsuited for edge environments, given that as previously mentioned in Section 1.1, for our goal, centralizing the computation (for example in data-centers) will eventually lead to a bottleneck for the system scalability. Furthermore, these types of systems are plagued with a single point of failure, making them unsuitable for volatile environments.

Unstructured resource location systems are attractive for systems that perform queries in search for resources with multiple copies or for range queries, however, this approach is inefficient when performing exact match queries, as a finding the exact resource in an unstructured resource location system requires flooding the entire system with messages.

Conversely, distributed hash tables are specially tailored towards exact match queries, but are less robust to churn and are subject to low-capacity nodes being a bottleneck in routing procedures.

2.4 Resource Monitoring

In this section, we will cover **resource monitoring**, which consists in tracking the state of certain aspects of a system, such as the device status, the capacity of links between devices, the status of available resources within a given geographical zone, among others, which is paramount for making effective management decisions regarding task allocations and managing the overlay network. However, if every node were to continuously collect, store and process the metrics of other nodes, the amount of communication and processing needed to do this would quickly overload the system. Consequently, there is the need to reduce the size of the data through a process called *aggregation*.

2.4.1 Aggregation

Aggregation consists in the determination of important, system-wide properties and it is an essential building block towards monitoring distributed systems [11] [31]. This technique can be employed, for example, towards computing the average of available computing resources in a certain part of the network or towards identifying application hotspots by aggregating the average resource usage in certain areas, among many other uses. There are two properties of aggregation functions: *decomposability* and *duplicate sensitiveness*.

	Decomposable		Non-Decomposable
	Self-decomposable		
Duplicate insensitive	Min, Max	Range	Distinct Count
Duplicate sensitive	Sum, Count	Average	Median, Mode

Table 2.2: Decomposability and duplicate sensitiveness of aggregation functions

Decomposability

A decomposable aggregation function is one where a function may be defined as a composition of other functions. Decomposable functions may be **self-decomposable**, where the aggregated value is the same for all possible combinations of all sub-multisets partitioned in the multiset. This happens whenever the applied function is commutative and associative (e.g. min, max, sum, count). A canonical example of a decomposable function that is not self-decomposable is average, which consists of the sum of all pairs divided by the count of peers that contributed to the aggregation. For non-decomposable aggregations, we need to involve all elements in the multiset. These are less desirable to perform in a large scale system, as the number of input values is large, and gathering all the input values may incur additional networking costs.

Duplicate sensitiveness

The second property of aggregation is **duplicate sensitiveness**, and it is related to whether a given value can or cannot occur several times in a multiset, as depending on the aggregation function, the presence of repeated values may influence the result. It is said that a function is **duplicate sensitive** if the result of the aggregation function is influenced by the repeated values (e.g. SUM). Conversely, if the aggregation function is **duplicate insensitive**, it can be successfully repeated any number of times to the same multiset without affecting the result (e.g. MIN and MAX).

Table 2.2 classifies popular aggregation functions in function of decomposability and duplicate sensitiveness as found in [31].

2.4.2 Aggregation techniques

In the following subsection, we provide context about the taxonomy of aggregation techniques:

Hierarchical aggregation

Tree-based approaches leverage directly on the decomposability of aggregation functions. Aggregations from this class depend on the existence of a hierarchical communication structure (e.g. a spanning tree) with one root (also called the sink node). Aggregations take place by splitting inputs into groups and aggregating values bottom-up in the hierarchy. Tree-based architectures also allow efficient multi-tree aggregation, which consists

in the calculation of an aggregation result through the exchange of partial averages data among all active nodes in the aggregation process [11].

Cluster-based techniques rely on clustering the nodes in the network according to a certain criterion (e.g. latency, energy efficiency). Then, within each cluster, a representative is responsible for local aggregation and for transmitting the results to other representatives.

Hierarchical approaches, due to taking advantage of device heterogeneity, are attractive in edge environments. However, due to the low computational power of devices, not all nodes may be able to handle the additional overhead of maintaining the hierarchical topology, furthermore, there are additinal concerns regarding failures when compared to ad-hoc aggregation.

Ad-hoc aggregation

Ad-hoc aggregation consists of a class of aggregation algorithms that calculate aggregations through periodic, randomized exchanges of messages. These types of algorithms allow an estimation of an aggregated value high accuracy while employing unstructured overlays [29], consequently, these retain the fault tolerancee and resilience to churn from these overlays.

2.4.3 Monitoring systems

Provided with this overview of aggregation techniques, we now discuss popular monitoring systems in the literature. For each system, we discuss what we believe to be their advantages and drawbacks as solutions for edge settings.

Astrolabe [48] is a distributed information management platform that aims at monitoring the dynamically changing state of a collection of distributed resources. It introduces a hierarchical architecture defined by zones, where a zone is recursively defined to be either a host or a set of non-overlapping zones. Each zone (minus the root zone) has a local identifier, which is unique within the zone where it is contained. Zones are globally identified by their *zone name*, which consists of the concatenation of all zone identifiers within the path from the root to the zone in question.

Associated with each zone there is a Management Information Base (MIB) containing attributes relative to that zone. These attributes are not directly writable, instead, they are generated by aggregation functions contained in special entries in the MIB. Leaf zones are the excepted from these restrictions, instead containing *virtual child zones* which are directly writable by devices within that virtual child zone.

The aggregation functions which produce the MIBs are contained in *aggregation function certificates* (AFCs). These contain a user-programmable SQL function, a timestamp and a digital signature. In addition to the function code, AFCs may contain other information, such as an *Information Request AFC*, that specifies which information to retrieve

from each participating host, and how to summarize the retrieved information. Alternatively, we may have a *configuration AFC*, used for specifying runtime parameters that applications may use for dynamic configuration.

Astrolabe employs gossip exchanges to update the MIBs, which provides an eventual consistency model: if updates cease to exist for a long enough time, all the elements of the system converge towards the same state. This is achieved by employing a gossip algorithm that selects another agent at random and exchanges zone state with it. If the agents are within the same zone, they exchange information relative to their zone. Conversely, if agents are in different zones, they exchange information relative to the zone which is their least common ancestor.

Not all nodes gossip information, within each zone, a node is elected (the authors do not specify how) to perform gossip on behalf of that zone. Additionally, nodes can represent nodes from other zones, in this case, nodes run one instance of the gossip protocol per represented zone, where the maximum number of zones a node can represent is bounded by the number of levels in the Astrolabe tree.

An agents' zone is defined by its system administrator, which is a potential limitation towards scalability, given that configuration errors have the potential of heavily raising system latency and reducing traffic locality. Additionally, the authors state that the size of gossip messages scales with the branching factor, often exceeding the maximum size of a UDP packet. Other limitations which arise from using Astrolabe are the high memory requirements per participant due to the high degree of replication, and the potential points of failure of the representatives of zones.

Ganglia [41] is a distributed monitoring system for high performance computing systems, namely clusters and grids. In short, Ganglia groups nodes in clusters, in each cluster, there are representative cluster nodes that federate devices and aggregate internal cluster state. Then, representatives aggregate information in a tree of point-to-point connections.

Ganglia relies on IP multicast to perform intra-cluster aggregation, it is mainly designed to monitor infrastructure monitoring data about machines in a high-performance computing cluster. Given this, its applicability is limited towards edge environments: (1) clusters are assumed to be in stable environments, which contrasts with the edge environment; (2) it relies on IP multicast, which has been proven not to hold in a number of cases; (3) has no mechanism to prevent network congestion; finally, (4) it requires manual configuration of the tree structure.

SDIMS [66] (Scalable Distributed Information Management System) proposes a combination of techniques employed in Astrolabe [48] and distributed hash tables (in this case, Pastry [51]). It is based on an abstraction that exposes the underlying **aggregation trees** provided by a DHT such as Pastry.

Given a key k, an **aggregation tree** is defined by the union of the routing paths from all nodes to the node responsible for key k, where each routing step along the path to k corresponds to a level in the aggregation tree. **Aggregation functions** are associated

with an attribute type and a name and rooted at *hash(attribute type, attribute name)*, which results in different attributes with the same function being aggregated along trees rooted in different parts of the DHT, enabling load-balancing.

This achieves communication and memory efficiency when compared to gossip-based approaches, because MIBs have a lesser degree of replication. However, as each node belongs to every aggregation tree, this could potentially hinder scalability in edge settings, given that low-capacity nodes may become overloaded if they are intermediate aggregation points in all aggregation trees.

Prometheus [47] is an open-source monitoring and alerting toolkit originally built for recording any purely numeric time series. We believe this tool is one of the most popular tools in the state-of-the-art in regard to querying and collecting multi-dimensional data collections. This solution uses a "pull" technique to aggregate metrics, which means it scrapes targets periodically to obtain its metric values. To do so, it requires a configuration file that dictates many aspects of its behaviour, such as the targets for scraping metric values, the periodicity at which to perform this scrape, how long to retain metrics in the database, among other aspects. Furthermore, Prometheus also allows the configuration of alarms that trigger (configurable) actions whenever a given criterion is met.

Finally, Prometheus also allows federation, which consists of a server scraping selected time-series from another Prometheus server. Federation is split in two categories, *hierarchical federation* and *cross-service federation*. In *hierarchical federation*, Prometheus servers are organized into a topology resembling a tree, where each server aggregates aggregated time-series data from a larger number of subordinated servers. Alternatively, *cross-service federation* enables scraping selected data from another service's Prometheus server to enable alerting and queries against both datasets within a single server.

2.4.4 Discussion

After the study of the literature related to monitoring systems, we believe there is a lack of monitoring systems targeted towards edge settings, as popular existing solutions often have centralized points of failure, rely on manual configuration or depend on techniques such as IP multicast, which make them unsuited for large-scale dynamic systems such as the ones found in edge environments.

Furthermore, we argue that large-scale monitoring systems purely based on distributed hash tables [66] are unsuitable for edge environments, provided these assume all nodes have an equal capacity, which we believe to mismatch the heterogeneity of edge environments. Other alternatives that better align with our objectives, such as Astrolabe [48] (given it can be configured with device heterogeneity in mind), require heavy amounts of message exchanges to keep information up-to-date and require manual configuration of the hierarchical tree, which is also be undesirable, provided the dynamicity of these environments.

2.5 Resource Management

In this section, we study resource management in the context of edge environments. Resource management consists in providing resources (e.g. computing power, memory, among others) to tenants (i.e. applications, frameworks, among others), such that these can perform their computations. In this section, we cover aspects of resource management solutions and study popular solutions in the literature.

2.5.1 Resource Management Taxonomy

A resource management system aims at controlling the distribution of resources among tenants. We may classify resource management architectures according to their *control* and *tenancy*.

2.5.1.1 Tenancy

The term tenancy in resource management refers to whether or not underlying hardware resources are shared among entities [24].

Single tenancy refers to an architecture in which a single instance of a software application and supporting infrastructure serves one customer. In single-tenancy architectures, a customer (tenant) has nearly full control over the customization of software and infrastructure.

Multi-tenancy consists of tenants sharing multiple resources across multiple processes and machines. This approach has clear advantages, as sharing the infrastructure leads to lower costs (e.g. electricity), and companies of all sizes like to share infrastructure in order to achieve lower operational costs.

However, providing performance guarantees and isolation in multi-tenant systems is extremely hard, resource management systems must avoid mismatching the resource allocation, as tenant-generated requests compete with each other and with the system generated tasks. Furthermore, tenant workload can change in unpredictable ways depending on the input workload, the workload of other tenants in the system, and the underlying topology.

2.5.1.2 Control

Control refers to how resource management systems allocates tasks to available resources, there are two alternatives towards performing resource allocations: either *centralized* or *decentralized*.

Centralized control consists in a centralized component with a global view of the state of the system making all decisions regarding resource allocations. Intuitively, given that a centralized component generates manages all the resources in the system, this component can easily enforce policies to achieve the desired performance guarantees or fairness goals

by identifying and only throttling the tenants or system activities responsible for resource bottlenecks [64].

Decentralized control architectures are defined by having the decision-making process regarding resource allocations distributed across multiple components [24]. This topic has yet not been subject to much research, although it is of extreme relevance towards edge environments. For example, if the system is globally distributed, it may take too long for a centralized controller to identify hotspots in a certain zone and load-balance them.

One of the key challenges in distributed resource management is ensuring that the components which perform resource assignments do not conflict with each other. Additionally, in a multi-tenant decentralized resource management system, tenants may request resources to different resource controllers in the system, and if they do not coordinate themselves, the application may be provisioned with too many (or too little) resources.

2.5.2 Resource Management Systems

Mesos [23] is a multi-tenant centralized resource sharing platform that attempts to provide fine-grained resource sharing within a data centre. The tenants for this platform are frameworks such as HDFS [5], MapReduce [13], among others, which in turn support multiple applications running within a DC. In short, the Mesos resource sharing system consists of a *master* process which manages *slave* daemons running on each cluster node. In order to achieve fault-tolerance for the master component, Mesos employs Zookeeper [25] to maintain replicas, elect a new master, and transfer state to a new master in case the active master fails.

The master implements fine-grained sharing of resources across frameworks by employing *resource offers*, which consist of lists containing free resources distributed among slaves. The master makes decisions about how many resources to offer to each framework, and the decision-making process is based on an arbitrary organizational policy, such as fair sharing or priority. Each framework that wishes to use Mesos must implement a *scheduler* and an *executor*. The scheduler registers with the Mesos master to receive resource offers, and the executor is the process that is launched on slave nodes to run the framework's tasks.

A limitation of the Mesos resource sharing platform is that it has limited scalability, given the central component issuing resource allocations (the original authors mention the system scales up to 50000 slave daemons on 99 physical machines), which is not enough for an edge environment. Furthermore, the resource offer model forces frameworks to employ a specific programming model based on schedulers and executors, which we believe to be too restrictive.

Yarn (Yet Another Resource Negotiator) [63] is a centralized multi-tenant resource sharing platform that decouples the programming model from the resource management

infrastructure and delegates many scheduling functions to per-application components. The architecture of YARN is composed by: a per-cluster Resource Manager (RM), multiple Application Masters (AM), and Node Managers (NM). The RM tracks resource usage and node liveness, enforces allocation invariants and arbitrates contention among tenants.

AMs run arbitrary user code, their duties in the system consist of managing the life-cycle aspects, including dynamically increasing and decreasing resource consumption, managing the flow of execution, and handling faults. Node Managers (NM) are worker daemons, whose responsibilities consist of managing container dependencies, monitoring their execution, and providing a set of services for them.

AMs send resource requests to the RM, containing the number of containers to request, the resources per container, locality preferences, and a priority level within the application. These requests are designed to capture the needs of applications while at the same time removing application concerns (such as task dependencies) from the scheduler. Because the RM is in charge of processing and scheduling all task distributions for each request made by AMs, it is effectively a *monolithic* scheduler. By consequence, there is a unique point of failure, which makes this system inadequate for large scale edge environments.

Omega [54] is a scheduler designed for grid computing systems composed by schedulers and workers. Each scheduler receives large amounts of jobs composed by either one or many tasks that have to be scheduled among workers. Contrary to YARN, which is monolithic, OMEGA uses multiple schedulers per cluster, each with a shared global view of the cluster state.

Schedulers make task placement decisions according to their view of the cluster state and their scheduling policy. If two or more schedulers attempt to schedule a task to the the same worker (i.e., generating a conflict), the worker first tries to accommodate both tasks, if it cant, it rejects the least important one.

One advantage of OMEGA in relation to MESOS is that MESOS resource attributions "lock" the resources to the corresponding framework, which means that only one framework is examining a resource at a time. While it achieves higher throughput in allocation operations, its main limitations are that: (1) in case the grid becomes overloaded, resource allocations can potentially start interfering with each other; (2) scheduling policies are harder to ensure; and finally, (3) all schedulers must have global knowledge of the system.

Edge NOde Resource Management [65] (ENORM) is framework aimed at employing edge resources towards applications by provisioning and auto-scaling edge node resources. ENORM proposes a three-tier architecture: (1) the Cloud tier, where application servers are hosted; (2) the middle tier, where the edge nodes are situated; and (3) the bottom tier, where user devices (e.g. smartphones, wearables, gadgets) are situated.

To enable the use of edge nodes, ENORM deploys a cloud server manager on each application server, which communicates with potential edge nodes, requesting computing services. Using these computing resources, it deploys partitioned servers on the edge nodes. Edge nodes are maintained in a global view.

ENORM authors tested the designed system using an online game inspired on Pokemon GO (iPokemon)[8]. The ENORM framework partitions the game server and sends user data to each edge node containing information regarding the users within that geographical location. Users from the relevant geographical zone then connect to the edge server and are serviced by a geographically closer edge node as if they were connected to the data centre. Limitations from this framework are the large size of the required information to perform the deployments, and similarly to previous solutions, the lack of fault-tolerance and scalability, from employing a centralized component to perform monitoring and management of resources.

FogTorch [6] is a service deployment framework aimed at determining eligible deployments for an application over a given Fog infrastructure, modeled by: (1) Cloud Data Centers, denoted by their location and software capabilities; (2) Fog Nodes, that consist of tuples containing: the location, hardware, the software capabilities, and the things directly reachable from the fog node; (3) Things, which are represented by a tuple denoting the thing (sensor or actuator) location and its type; (4) QoS profiles, that are sets of QoS profiles composed by the latency and bandwidth of a communication link. (5) Applications, which are composed of independent sets of components, each with a set of requirements regarding QoS profiles, hardware and software capabilities, and things. Then, authors model service deployments as restrictions over the system model and employ a greedy heuristic, which reduces the search space of devices constituting options for these service deployments.

FogTorch is also the base for **FogTorchPI** [7], which is a solution that employs the system model of FogTorch, however instead of a greedy approach, it uses Monte Carlo simulations to calculate the best possible deployment configurations.

These solutions provide a comprehensive system model which models many different types of application requirements, however,similarly to FogTorch, it requires an updated global view of the system, which requires collecting a large amount of information to a central entity, limiting system scalability.

2.5.3 Discussion

Although resource management systems have been present for many years, these are often tailored towards small scale environments composed by homogenous devices in stable environments, which contrast with the edge of the network, where devices are extremely numerous, operate on a decentralized fashion, and are highly heterogenous.

We argue that a centralized controller is not an ideal solution for an edge environment, given the fact that as the number of devices in the system increases, so does the number of resources to track, and the harder it is for a centralized component to have an up-to-date global view of the system.

Due to their low capacity, devices at the edge of the network are very susceptible to workload changes, for example, a 5G tower that is hosting services cannot handle a

drastic increase in the number of users it is serving. In this scenario, we argue that in order to maintain pre-established performance criteria, devices must autonomously make resource management decisions such as scaling an allocation horizontally or vertically in order to quickly meet the demands of users/tenants.

2.6 Summary

The purpose of this chapter was to provide a brief overview of the studied relevant works and techniques found in the literature regarding (1) the edge environment and execution environments for edge environments; (2) construction of overlay networks; (3) resource monitoring platforms, and (4) resource location systems, with emphasis on analyzing their applicability toward edge Environments. Firstly, we began by studying the devices that we believe compose these environments and debated the applicability of popular execution environments for edge-enabled applications, following we addressed popular architectures and implementations of both structured and unstructured overlay networks, and analyzed popular techniques in the literature used towards performing resource location and discovery in these networks. After this, we examined related work regarding collecting metrics in a decentralized manner.

In the next chapters, we present the proposed solution, that we named DEMMON, which draws inspiration from the study of the state-of-the-art to enable the decentralized management and monitoring of resources at both data centres and the edge of the network.

3

GO-Babel

The first contribution of this thesis is an event-based framework called GO-Babel, available on [44]. This framework is a port in Golang [18] of Babel [1] with additions focused on fault detection and latency probing. Babel, in turn, is inspired on the model proposed by Yggdrasil [11].

The decision to build this framework arose from the need to use Babel for building the distributed protocols and the decision to use Golang during this dissertation (due to its primitives for building concurrent systems). Given that there was no implementation of Babel in Golang, and the current Babel implementation lacked some needed features such as a fault detector and a latency measurement tool, we implemented a new version in Golang with these additions.

3.1 Overview

In summary, this framework has the following main objectives:

1. Abstract the networking layer, providing **channels**, which are essentially an abstraction over TCP connections, providing callbacks whenever outbound or inbound connections are established or terminated and whenever messages or sent or received from the respective operating system buffers.

2. Execute protocols in a single-threaded environment and provide abstractions inter-protocol communications, such as request-reply and notification patterns.

3. Provide abstractions for handling and managing time-based events (timers).

4. Provide a layer of abstraction over node latency probing and fault detection.

In Figure 3.1 we provide a high-level overview of the architecture of this framework, composed of five main components that communicate via callbacks. We now summarize each components' roles within the framework:

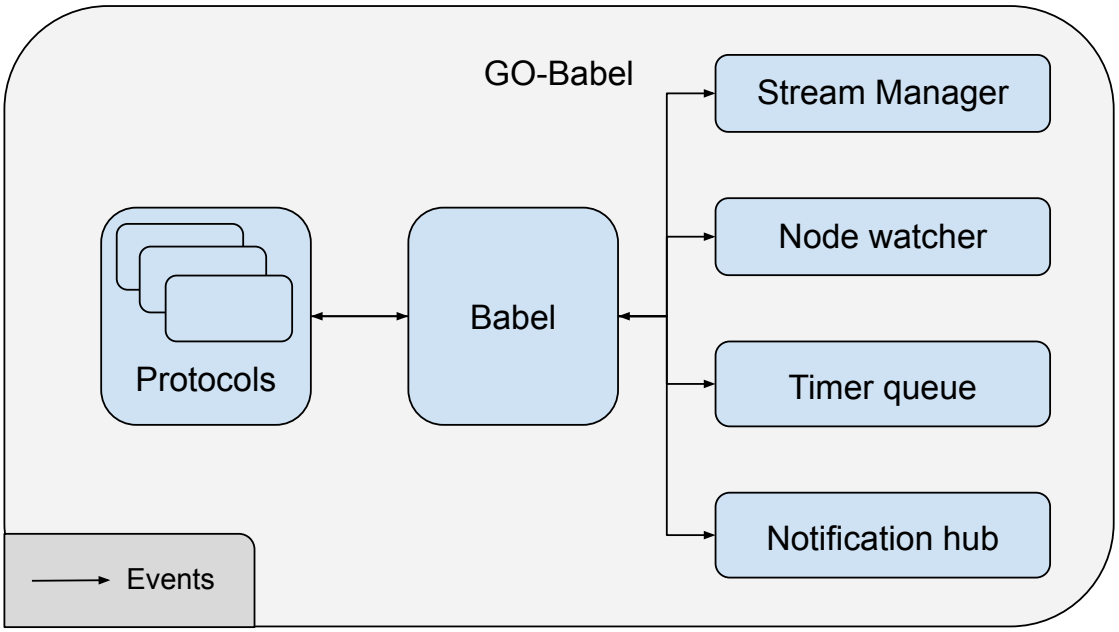

Figure 3.1: An overview of the architecture of GO-Babel

1. Babel is the component tasked with initializing the protocols and all the other components according to issued configurations. It also acts as a mediator between the protocols and the remaining components.

2. The Stream Manager is responsible for connecting to new peers, handling incoming/outbound connections, and sending messages through these connections. Interactions with this component are done through requests to, for example, dial a new node, or send a message through an established connection. When these requests are complete, the Stream Manager emits a notification back to the protocol that issued the request. The Stream Manager also provides operations for sending messages in temporary connections (either using TCP or UDP).

3. The Timer Queue allows the creation and cancellation of timers and manages the lifecycle of timers issued by the protocols, delivering events to protocols whenever timers reach their expiry time. Two types of timers are allowed, the first are single-trigger timers, which only trigger once, and then are disposed of, the second type of timers periodic timers, which trigger at the set periodicity until cancelled.

4. The Notification Hub is responsible for handling notifications and notification subscriptions, allowing protocols to substribe to certain notifications IDS. Any protocol subscribed to a notification ID always receives an event whenever another protocol emits that notification.

5. The Node Watcher is the new addition to the framework. It allows for protocols to collect metrics regarding the latency and the current status (failed or running) of another node in the system. An explanation of this component in further detail is provided in the following section.

As previously mentioned, the Node Watcher is the only new addition to the framework, and consequently, it is the component explained in further detail. The remaining components of this framework were implemented similarly to the equivalent components of Babel [1] and Yggdrasil [11].

3.2 Node Watcher

The motivation to build this component was a lack of tools to measure latency in the original design of Babel. If, for example, a protocol were to measure the latency to a node without an active connection, it would need to establish a new TCP connection and use it to send the probes. In this case, both the fault detector and latency detector logic are in the protocol, which is sub-optimal since the same logic would have to be replicated by any protocol that wishes to optimize its active connections using latency as a heuristic. Alternatively, if a protocol measures latencies in a separate module asynchronously (making the code reusable), this would break the single-threaded nature of the execution of protocols in Babel (which impacts usability), and protocols would have to deal with race conditions of altering the state concurrently. Due to this, we believe that encapsulating this logic in an optional component and expose it in a Babel-compatible interface is the preferred option, which was the one used.

The Node Watcher is an optional component that, if registered, it will listen for probes in a custom port (specified in the configuration parameters) and send a reply with a copy of the contents back to the original senders. These probes are sent via UDP and carry a timestamp used by the original sender to calculate the round-trip time to the target node.

The main interface for the Node Watcher is composed of two functions, "watch" and "unwatch". When a node is "watched", the Node Watcher starts sending probes to the target node according to the issued configuration settings and instantiates a PHI-accrual fault detector [22] together with a rolling-average latency calculator for that node. When the node receives replies with copies of sent probes, it updates the corresponding rolling average calculator and fault detector. Conversely, when a node is "unwatched", the Node Watcher stops issuing the probes and deletes the fault detector and latency calculator.

When a protocol issues a command to watch a node, if the "watched" node fails to reply within a time frame, the Node Watcher falls back to TCP. This fallback aims to overcome cases where the watched node may be dropping UDP packets due to a constraint in its infrastructure (i.e., a firewall rule). If the watched node also does not accept the TCP connection, the Node Watcher sends a notification to the issuing protocol informing it failed to "watch" the node.

In order to prevent protocols from having to set timers to check the nodes' latency calculator or fault detector, the Node Watcher also allows the possibility of registering "observer" functions (or conditions), which return a boolean value based on the current node information. The Node Watcher then executes these functions periodically, and if one returns true, a notification gets sent to the issuing protocol. In order to prevent protocols from getting overloaded with notifications when a condition returns "true", these may configure a grace period, which the Node Watcher will wait for until re-evaluating the condition.

3.3 Summary

We believe Go-Babel is a small, yet valuable contribution to the distributed systems community not only as it provides more choice for developers in terms of choice of programming languages when choosing a distributed systems framework, but also because it provides many abstractions which ease thee deployment of self-improving protocols that employ latency as an optimization heuristic. Finally, it also provides a secondary fault detector which may be employed alongside the implicit fault detector granted by TCP connections, which is useful for cases when the TCP connection has failed yet the nodes did not receive any errors (e.g. when the ethernet cable is cut). Lastly, as the implementation is in Golang [18], it allows easier integration with a range of packages already implemented in the language, many of which are useful for combining with GO-Babel.

4

DeMMON

DeMMon (Decentralized Management and Monitoring framework) is a monitoring framework that aims to tackle the needs of decentralized resource management tools. These tools, as previously mentioned, must perform resource management decisions, such as load balancing or QOS optimizations, supported by partial and localized knowledge of the system. It is the goal of this framework, through the on-demand decentralized collection, aggregation, and storage of metrics in the form of time-series, to provide this knowledge base. We now detail what we believe to be the most common requirements of such tools:

1. **Locality, by interacting with a partial set of nodes from the system**, optimized according to a certain proximity heuristic. This set is crucial such that a certain node has others to interact with to perform the aforementioned localized resource management decisions. In our framework, we chose latency as the heuristic for the proximity heuristic. The reasons for this choice were that not only does it does not rely on external tools, such as traceroute or a reverse IP-to-geolocation service, nor does it require pre-configuration of geolocation, making it possible for all nodes' configurations to be similar (thus making the deployment of large quantities of nodes easier).

2. **Storage and querying of metric values**. As it is impossible to know ahead of time what type of information resource management systems and the functions to aggregate that information would otherwise require, we also believe that it is a requirement to **be as flexible as possible regarding metrics types and aggregation functions**. Furthermore, by allowing resource management systems to create custom-tailored metric formats tailored for their own needs, we believe it may even promote higher efficiency, as this feature may prevent inefficient workarounds from metric type restrictions.

3. Ensure there are ways to **obtain the globally aggregate value of a metric distributed across one or more nodes in the system**, for example, the total number

of nodes, service replicas, among others, without having to rely on a central component. This feature is important for resource management tools to, for example, maintain a (configurable) ratio of service replicas to nodes: by simultaneously collecting both the number of nodes in the system and the number of replicas, nodes can perform local decisions such as creating or decommissioning replicas, whenever the desired ratio of reaches a certain bound. Or alternatively, for example, for periodically collecting the number of nodes in the system to act as a configuration parameter for other systems.

4. Have a way to **obtain the aggregate value from a set of "nearby" nodes**. This feature is useful for decentralized resource management systems as it allows them to perform actions in a decentralized manner: by collecting the metrics relative to the usage of nearby nodes, each node may decide (e.g to improve a service's latency through proximity, to or reduce the load on a saturated service) to replicate or migrate service, motivated by this partial aggregate value.

5. **Have a way to collect non-aggregated metric values from a set of "nearby" nodes**. Similar to item 4, resource management frameworks may need to collect non-aggregated values to perform actions. In a service deployment context, it may want to collect the geographical positions of some nodes and deploy service replicas nearer to the current service clients' location.

6. Provide ways to efficiently **propagate information** across nodes in the system. This is useful for resource management systems, as it prevents the overhead of establishing information propagation at the resource management layer.

7. Ensure ways to **receive notifications based on issued alerts** that trigger whenever a supplied condition is met. This prevents clients of this system from resorting to periodically requesting/consulting information and performing the verifications themselves, saving unnecessary computation. By setting these alarms, resource management tools can, in turn, trigger resource management actions, for example, set an alarm that triggers if the mean of the CPU usage over the last N seconds reaches a certain threshold. When this alarm triggers, perform load-balancing or service migrations to spread the CPU load throughout nearby nodes. Furthermore, it is important to note that it is possible to create alerts on aggregated metric values.

Having enumerated what we believe to be the requirements of such tools, we now provide a brief overview of the devised framework, which aims to fulfill these requirements.

4.1 Framework overview

The devised framework (illustrated in Figure 4.1) is coalesced by four main modules: the overlay network, the aggregation protocol, the API, and the monitoring module. In the following paragraphs, we describe each module's role within the framework and how they contribute to fulfilling the above-mentioned requirements.

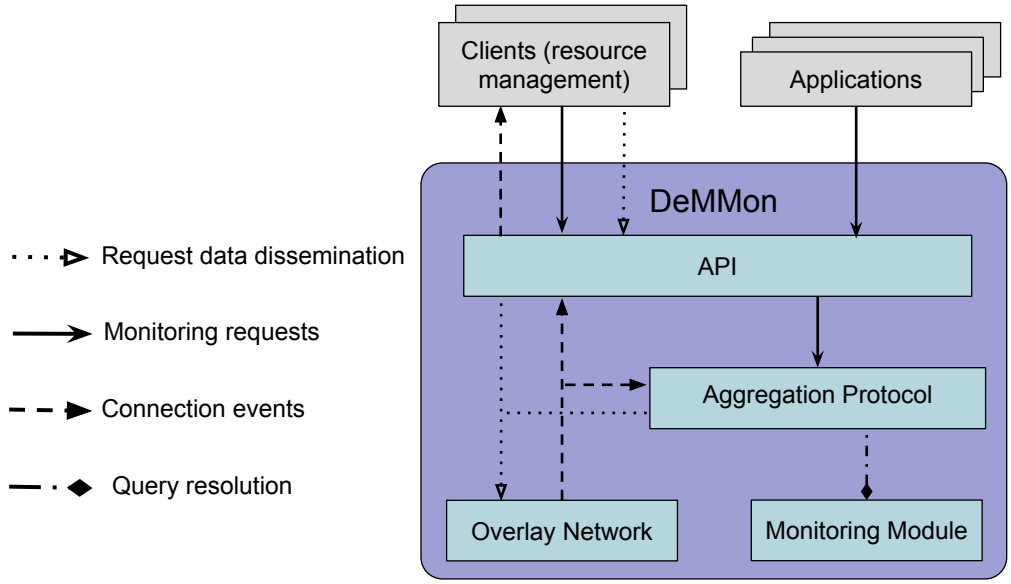

Figure 4.1: An overview of the architecture of DeMMon

First, the **API** exposes the functionality of the framework, its main objectives are to (1) allow resource management solutions to collect metrics about nodes (or services they host) in the system; (2) allow those metrics to be queried through the use of a query language; (3) allow registering alarms which trigger based on conditions which evaluate the collected information. It is important to notice that the API is not the component tasked with gathering the information to perform these tasks. Instead, it exposes the results and mediates the interactions between the clients and the remaining modules.

Second, the **monitoring module** is tasked with storing metrics, resolving queries regarding stored metrics, removing expired metrics, periodically evaluating registered alarms, and triggering callbacks which the API then propagates to the client. This module satisfies points 2 and 7 of the aforementioned requirements.

The **overlay network** is responsible for building a latency-aware multi-tree-shaped network. Nodes in this network use latency, node capacity, and a set of logical rules to change their location either from one tree to another or within their tree until they have

an optimized set of nodes (according to latency). The connections resulting from the operation of this protocol are the basis for the aggregation protocol. In addition, this module also offers limited horizon flood techniques, exposed through the API, fulfilling the points 1 and 6 of the requirements presented previously.

Finally, the **aggregation protocol** is a component that performs on-demand metric collection based on issued commands from the API. This component takes advantage of the overlay networks' established connections and hierarchical structure to perform efficient distributed aggregations. It allows three types of decentralized aggregation: (1) *tree aggregation*, which consists of collecting metrics and merging them using the overlay protocols' trees, collecting a globally aggregated value in the tree roots (or a partial view of the system for nodes that are not the root of the overlay); (2) *global aggregation*, where nodes also use their tree connections to efficiently collect a globally aggregated value (independently of being the root of the tree); and (3) *neighbourhood aggregation*, where nodes collect values (non aggregated) of nearby nodes in term of hop proximity. These three mechanisms satisfy points 3, 4 and 5 of the aforementioned requirements.

In the following sections, we will provide a detailed explanation of each individual modules' design and implementation, starting by the **overlay network** (section 4.2), followed by **aggregation protocol** (section 4.3), and lastly, the **monitoring module** (section 4.4) and **API** (section 4.5).

4.2 Overlay network

In this section, we discuss the design of the devised overlay network protocol, which aims to build and maintain a latency and capacity-aware tree-shaped network, where capacity represents one, or a combination of, values that denote the node's networking or computing capacity (in our case, we used only bandwidth as each nodes' capacity). We begin by providing the considered system model, then follow with an overview of the mechanisms responsible for building and maintaining the tree, and lastly, we conclude the chapter with a summary and discussion of the protocol.

4.2.1 System Model

The considered system model for this protocol is a distributed scenario composed of nodes connected to the Internet and set up to enable the reception or emission of messages via the Internet (either with an external IP or port-forwarding). We also assume that nodes are spread throughout a large geographical area, have varied capacity values, and are distributed among both Cloud and Edge settings.

Regarding the fault model, we assume that all but a small portion of nodes (also known as the landmarks, which in our model represent data centres) can fail, and when other nodes fail, they do so in a crash-fault manner, stopping the execution of the process along with all emissions and receptions of messages at the time of failure. We assume

landmarks have additional fault tolerance provided their privileged infrastructure, or alternatively, we assume that other mechanisms such as replication could be employed to ensure that faulty landmarks are swiftly replaced in case of failure.

Finally, all nodes must run the same software stack with similar configuration settings, installed a priori.

4.2.2 Overview

As previously mentioned, the main objective of the devised protocol is to establish a latency and capacity-aware multi-tree-shaped overlay network, rooted in the previously mentioned landmarks. The motivations for the choice of a tree structure for our network are the following: (1) to map the cloud-edge environment, by rooting the trees on nodes running DCs in the cloud, and setting up the remaining nodes with less capacity in positions where they can be coordinated from the roots (2) to be able to map the heterogeneity of each device in the environment: by biasing the placement of nodes in the tree such that nodes with higher capacity are placed higher in the tree, and nodes with lower capacity are biased towards lower levels of the tree, nodes are used more or less according to their capacity values; (3) the tree structure can be easily employed to perform efficient aggregations, by propagating and merging values (recursively) from the lower to the higher levels of the tree, which is the basis for the aggregation protocol presented in Section 4.3.1; and finally, (4) by leveraging on the tree structure, nodes can propagate information efficiently, given that, in a network composed of N nodes, broadcasts require only N-1 message transmissions to reach all nodes in the network.

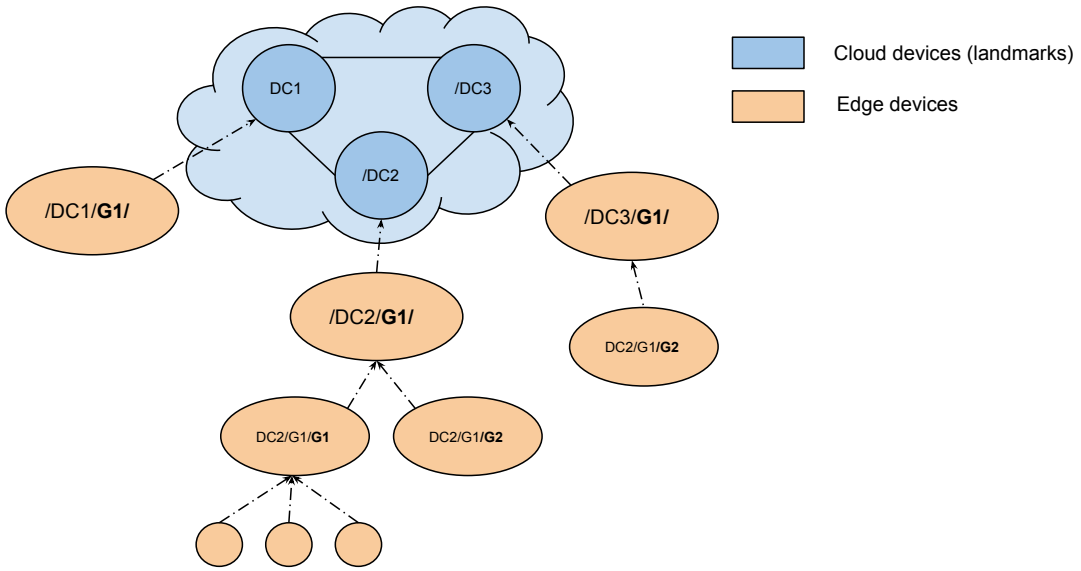

Figure 4.2: An example of a network established by the devised protocol (with 3 landmarks)

The type of tree structure our protocol aims to establish and maintain is represented in Figure 4.2, which, as previously referenced, is composed of multiple interconnected trees. As the reader can note, every node has attributed an identifier, which is the concatenation of the parents' ID with an assigned ID (this mechanism is explained in further detail in Section 4.2.2.3). The nodes connected to the landmarks (which we refer to as their **children**) may themselves be the **parent** of their own children, which would have the landmark as their **grandparent** (which is the case of "/DC2/G1/G2" and "/DC2/G1/G1"). Intuitively, the **descendants** of a node are all of its children and children's children, recursively, until the leave nodes. All nodes which share the same parent (**siblings**) are connected among themselves, forming a **group**, whose size is biased (but not guaranteed) to be within two configurable upper and lower bounds. Therefore, all nodes have active connections to their parent, children and siblings. The combination of a node's active connections may be called its **active view** (following the nomenclature introduced in [32]).

The devised algorithm is composed of three main mechanisms: (1) the **join** mechanism, which aims to establish the initial tree structures, (2) the **active view maintenance**, responsible for biasing the number of connections for each node and optimizing the connections of each node, (3) and finally **passive view maintenance**, responsible for collecting information about peers which are not in the active view, which are used for both fault tolerance and connection optimizations.

4.2.2.1 Join mechanism

The Join mechanism is the mechanism responsible for establishing an initial parent connection. It is the aim of this mechanism (from the joining node standpoint), to establish a connection with the node with the least cost possible. This mechanism is the first to be executed by all nodes in the system, and it essentially consists of a depth-first search in the established DeMMon trees. The pseudocode for this algorithm can be observed in Algorithm 1.

The first step of the algorithm (line 4) is to initialize the state of the joining node, that is materialized by: (1) a map called *contactedNodes* of type "Node", containing all nodes contacted successfully in the join process (indexed by a string representation of their IP), (2) a collection named *nodesToContact* of type "Node" containing the nodes to yet to contact in the join process, (3) a map of timer IDS indexed by strings, containing the timer IDS for each contacted node, named *joinTimeouts*, (4) a variable called *prevBestP* containing the best (lowest latency) node contacted so far in the join process, (5) a variable named *joinReTimeoutId*, containing a timer id for a timer used as a timeout for the chosen node in the join process, (6) a variable of type "Node" denoting the peer executing the protocol, and finally (7), a set containing the landmarks of the network, named *landmarks*. The type "Node" is a collection of attributes regarding a certain physical node, composed of: (i) latency measured, (ii) its current parent, (iii) number of children, (iv) whether the

Algorithm 1 Join Protocol

```
1:  Types
2:      Node : <lat, parentIP, nrChildren, replied, IP, ID, coords, version, children<IP, nrChildren
3:
4:  State
5:      contactedNodes                                  ▷ collection of all successfully contacted nodes
6:      nodesToContact set<Node>                        ▷ nodes being contacted
7:      joinTimeouts : dict<Node, time>                 ▷ collection of contacted nodes -> timerIDs
8:      bestPeerLastLevel : Node                        ▷ the best peer contacted so far in the join process
9:      joinReqTimeoutTid : string                      ▷ timerID for join messages
10:     prevBestP : Node                                ▷ myself
11:     landmarks : set<IP>                             ▷ landmark nodes
12:
13: Upon Init(landmarks : set<IP>, selfIP, isLandmark) Do
14:     landmarks ⟵ landmarks
15:     joinTimeouts, prevBestP ⟵ {}, nil
16:     if isLandmark then addLandmarkUntilSuccess(landmarks)
17:     else contactNodes(landmarks)
18:
19: Upon receive(Join<>,sender) Do
20:     sendMessageSideChannel(JoinReply<self.parent, self.node, self.children>, sender)
21:
22: Upon receive JoinReply(<parentIP, node, children>, sender) && measuredLatency(lat) Do
23:     if node.IP ∈ nodesToContact then
24:         if parentIP ∈ Landmarks then
25:             self.coordinates[getIdx(landmarks, sender)] = lat
26:         nodesToContact[node.IP].lat ⟵ lat
27:         nodesToContact[node.IP].children ⟵ children
28:         nodesToContact[node.IP].parent ⟵ parentIP
29:         nodesToContact[node.IP].replied ⟵ true
30:         cancelTimer(joinTimeouts[sender])
31:         delete(joinTimeouts, sender)
32:     else
33:         nodesToContact.delete(node)
34:
35: Upon (forall n ∈ nodesToContact -> n.replied) Do
36:     contactedNodes.appendAll(nodesToContact)
37:     for node in sortedByLatency(nodesToContact) do
38:         if (node.IP ∉ landmarks) && node.nrChildren == 0 then
39:             continue                                ▷ check if node has enough children
40:         if prevBestP != nil && (prevBestP.lat ≤ node.lat || prevBestP.nrChildren < config.minGroupSize) then
41:             joinAsChild(prevBestP)
42:         else
43:             prevBestP ⟵ node
44:             toContact ⟵ [c ∈ prevBestP.children -> c.nrChildren > 0]
45:             contactNodes([c.IP for c in toContact])
46:         return
47:     if prevBestP != nil then joinAsChild(prevBestP)
48:     else abortJoinAndRetryLater()
49:     return
50:
51: Upon JoinTimeoutTimer(node) || NodeMeasuringFailed(node) Do
52:     if (L in Landmarks) then abortJoinAndRetryLater()
53:     else delete(nodesToContact[L])
54:
55: Upon JoinRequestTimer(p : Node) Do
56:     if sender == prevBestP then
57:         if p.parentIP != nil then
58:             prevBestP ⟵ contactedNodes[p.parentIP]
59:             joinAsChild(prevBestP)
60:         else
61:             abortJoinAndRetryLater()
62:
63: Upon receive(JoinRequest<>, sender) Do
64:     childID ⟵ addChildren(sender)                   ▷ new chilren is established, and an ID is generated for it
65:     sendMessageSideChannel(JoinRequestReply<childID, self>, sender)
66:
67: Upon receive(JoinRequestReply<myID, parent>, sender) Do
68:     if sender == prevBestP then
69:         parent ⟵ sender                             ▷ Adds Parent is established, join complete
70:         cancelTimer(joinReqTimeoutTid)
71:         self.ID ⟵ parent.ID + "/"+ myID             ▷ Later used in shuffle mechanism
72:
73: Procedure joinAsChild(p : Node)
74:     joinReqTimeoutTid ⟵ setupTimer(JoinRequestTimer<p>, config.JoinTimeout)
75:     sendMessageSideChannel(JoinRequest<>, p.IP)
76:
77: Procedure contactNodes(ips : IP[])
78:     nodesToContact ⟵ {}
79:     toContact ⟵ [Node<0,nil,0,false,ip,false,[]> for ip in ips]
80:     for n in toContact do
81:         nodesToContact[n] ⟵ n
82:         MeasureNode(n)
83:         sendMessageSideChannel(JoinMessage<>, n)
84:         joinTimeouts[n] ⟵ ⟵ setupTimer(JoinTimeoutTimer(n), config.JoinTimeout)
85:
```

node replied to the message, (v) its IP, (vi) an array of coordinates (denoting its measured latency to each landmark, used in passive view maintenance mechanism), and finally, (vii) an array of its childrens' IP and their respective number of children.

The procedures that are used to join the tree differ regarding if the node is a landmark or not (which is a configuration parameters provided at the setup of a node): in the case of landmarks, these attempt to repeatedly establish a connection with other landmarks through the emission of a special message. Landmarks that receive this message always send a reply and establish a connection to the sender of the message (line 16). Any joining landmark only stops sending messages to other landmarks when the respective reply is received and an outgoing connection is established.

Nodes that are not landmarks begin the process of finding their initial parent in the DeMMon tree. This process is initiated by measuring the current latency and sending a JOIN message (via a temporary TCP channel) to the landmarks. For each message sent, a timer is created, and its ID is stored in the *joinTimeouts* map (line 17). Whenever a node receives this JOIN message, it sends a JOINREPLY message back to the original sender containing: its parent, itself, and its children (line 19).

During the wait process, the joining node waits for either the responses from the contacted nodes, for any timer in the *joinTimeouts* map to trigger, or for any failed latency measurements. In the second and third cases, the contacted node is excluded from the join process, and it is resumed as normal (line 51). In case there are no nodes left to resume the join process, or if the excluded node is a landmark, then the node waits a configurable amount of time until attempting to re-join the overlay again.

If the contacted node has not failed, and the joining node receives the JOINREPLY (line 22), it checks if it came from a timed-out node or from any node whose parent was not contacted in the join process (e.g. if the contacted node changed parent during the join process), if any of these situations occurs, then the message is discarded. If none of these situations occurs, the message is not discarded, and the information contained in the JOINREPLY message is stored in the *contactedNodes* map.

Whenever the joining node has either received the JOINREPLY messages from all contacted nodes or they have been excluded from the join process, it evaluates all the successfully contacted nodes attempting to find the contacted node with the lowest latency that is a suitable parent (by suitable, we mean a node that already has children or is a landmark). This procedure is performed by sorting the nodes in ascending order of measured latency and performing the following verifications:

1. Verify if the node already has any children or if the node is a landmark (landmarks can become parents of any node, except other landmarks) (line 38). If it is neither of these situations, then the node is excluded from the join process.

2. Verify if there was a node already contacted previously which was a suitable parent and had lower measured latency. In case there was, the joining node sends a JOIN-REQUEST message (requesting to be its child), sets up a *JoinRequestTimer* for the

lower latency node, and stops the join process. (line 40)

3. Verify if the current node has both enough children and has the lower latency when compared to the best previous last node. If so, then the joining node assigns it as its best node so far and starts a new recursive step by sending JOIN messages and measuring the latency to the children of that node which themselves have more than one children (line 43). Note that if none of the current nodes' children is a suitable parent (i.e. have no children themselves), then the condition in line 35 is triggered, and the joining node requests the current best node to be its parent.

4. If none of the verified peers was suitable to start a new recursive step (either because it had no children or had higher latency when compared to a previously contacted node), then the node joining node sends a JoinRequest to the best previously contacted node (in terms of latency), and sets up a *JoinRequestTimer* for it (line 48).

The node that receives this JoinRequest message replies with a *JoinRequestReply* and adds the node to its children by attempting to establish an outbound connection to it. Then, the join process is concluded with the reception of the *JoinRequestReply* and the establishment of the connections between the two nodes. If, however, the *JoinRequestTimer* timer triggers while waiting for the response, the node will fall back to the parent of the selected node, (and do so recursively, in case the parent of the current node failed as well, until reaching the landmark of that branch).

4.2.2.2 Active view maintenance

The second mechanism of the devised membership algorithm, called active view maintenance, is the mechanism responsible for maintaining the size of the groups and optimizing the nodes' active connections (when possible). This mechanism is performed by each parent periodically when their group size is above a certain threshold size, through the emission of messages to one (or more) of its children, proposing they should connect to another provided parent. The nodes chosen to be the new parents are chosen using latency measurements and node capacity as heuristics, obtained via periodic transmission from every child to their parent.

The pseudocode for this mechanism is presentend in Algorithm 2, and it starts by defining the necessary state to execute it, starting by the nodes' active view (parent, children, and siblings), and an auxiliary map of sets, named *childrenLatencies*, which holds the latencies of each children to every other children. (lines 2-5).

This mechanism starts with the periodic propagation of information from the parent to its children and vice versa. As denoted in lines 7-16, each parent transmits to its children a list of its current children (the siblings, from the children's point of view). Then, the children nodes measure their latencies to each of their siblings, and propagate the obtained latency values back to the parent. When this information is received (lines 17 and 25), it is merged into their respective local view for later use.

Algorithm 2 Membership protocol (Active view Optimization)

```
1:  State
2:      parent : Node                                                          ▷ defined in join
3:      children : dict<string,Node>                                           ▷ defined in join
4:      siblings : dict<string,Node>
5:      childrenLatencies : dict<string:dict<string:number»   ▷ Holds the latencies of each children to every other children
6:
7:  Every config.updatePeriodicity Do
8:      if parent != nil then
9:          sLatencies ⟵ set()
10:         for sibling in siblings do
11:             sLatencies.append(<sibling.IP,sibling.measuredLatency)
12:         sendMessage(UpdateChildStatus<children, sLatencies>, parent)
13:     for child in chidren do
14:         sendMessage(UpdateParentStatus<self, chidren
15: child>)
16:
17: Upon receive(UpdateParentStatus<parent, children>, sender) Do
18:     if sender == parent.IP then
19:         parent ⟵ parent
20:         self.ID ⟵ parent.ID + "/"+ myID
21:         grandParent ⟵ grandParent
22:         siblings ⟵ siblings
23:         measureSiblingLatency(siblings)
24:
25: Upon receive(UpdateChildStatus<child, childSiblingLatencies>, sender) Do
26:     if children[sender] != nil then
27:         children[sender] ⟵ child
28:         childrenLatencies[sender] ⟵ childSiblingLatencies
29:
30: Every config.evalGroupSize Do
31:     if len(children) <= config.maxGroupSize then
32:         return
33:     childrenLatValues ⟵ set()
34:     for c1 in children do
35:         for <c2, lat> in childrenLatencies[c1] do
36:             if lat - c1.measuredLatency > d.config.maxLatDowngrade then
37:                 continue
38:             if c1.cap > c2.cap then childrenLatValues.add(<c1,c2,lat>)
39:             else childrenLatValues.add(<c2,c1,lat>)
40:     kickedNodes, newParents ⟵ set(), set()
41:     pChildren ⟵ dict<string,set<Node>>                  ▷ set of potential children for each children
42:     sortByLatency(childrenLatValues)
43:     idealGroupSize ⟵ config.maxGroupSize - config.MinGroupSize
44:     for <c1,c2,lat> in childrenLatValues do
45:         if len(children) - len(kickedNodes) <= config.maxGroupSize then
46:             break
47:         if c1 ∈ kickedNodes || c2 ∈ kickedNodes || c2 ∈ newParents then
48:             continue
49:         if c1.nrChildren == 0 && newParents c ∈ 1 then                  ▷ Node is not yet a parent
50:             pChildren[c1] ⟵ pChildren[c1] ∪ c2
51:             if len(pChildren) == config.MinGroupSize then
52:                 for potentialChild in pChildren[c1] do
53:                     kickedNodes ⟵ kickedNodes ∪ potentialChild
54:                     send(OptimizationPropose<c1>, potentialChild)
55:                 for <nIP,pontentialChildrenTmp> in pChildren do
56:                     pontentialChildrenTmp.deleteAll(pChildren[c1])
57:                 pChildren[c1] ⟵ set<Node>
58:                 newParents ⟵ newParents ∪ c1
59:         else
60:             kickedNodes ⟵ kickedNodes ∪ c2
61:             for <nIP,pontentialChildren> in pChildren do
62:                 pontentialChildren.delete(c2)
63:             send(OptimizationPropose<c1>, c2)
64:
65: Upon receive(OptimizationPropose<newParent>, sender) Do
66:     if sender == parent then
67:         send(OptimizationProposeRequest<sender>, newParent)
68:
69: Upon receive(OptimizationProposeRequest<p>, sender) Do
70:     if  p == parent && sender in siblings then          ▷ parent issuing the message is my parent
71:         addChildren(sender)
72:         send(OptimizationProposeRequestReply<true,p>, sender)
73:     else
74:         sendMessageSideChannel(OptimizationProposeRequestReply<false,p>, sender)
75:
76: Upon receive(OptimizationProposeRequestReply<reply,p>, sender) Do
77:     if parent == p then
78:         if reply then
79:             sendMessageAndDisconnectFrom(DisconnectMessage<>, parent)
80:             addParent(sender)
81:     else
82:         sendMessageSideChannel(DisconnectMessage<>, p)
83:
```

The second part of this mechanism is responsible for maintaining the group sizes by creating new parents or by sending children to already created groups (line 30). This mechanism is triggered periodically but only executes if the number of children of a node (denoted the *proposer* node) exceeds the configured maximum number of children per parent. In this mechanism, a *proposer* node proposes to one of its children (denoted node *C1*) a change of parent to one of the proposers' children (denoted the *C2* node).

When triggered, the proposer node begins by merging all of its received latency pairs into a single set, where the node with the highest capacity is the first node of each pair. While doing so, it discards any new edges which would otherwise lower the overall latency of the system by a larger than configured amount (lines 34-39). Then, the *proposer* node iterates over the merged edge pairs set by ascending order of latency cost, performing the following steps:

1. If the number of current children minus the number of nodes already sent to lower levels is lower than the configured maximum group size, then the mechanism has achieved its purpose, and the *proposer* node concludes the mechanism (line 45)

2. If any of the two nodes were already sent to lower levels of the tree in previous steps, then the current edge is skipped (line 47).

3. Then, if the node with higher capacity of the edge pair has no children yet, the lower capacity node is added to its *possibleChildren* set (line 50). When this set has the same size as the minimum configured group size, then the node issues OptimizationPropose messages for each node of the *possibleChildren* set, and removes each child from every other node's potential children (lines 52-57). Alternatively, if the higher capacity node already is a parent (either because some nodes were already chosen to form its group, or because it was already a parent previously), then the coordinator node issues an OptimizationPropose message to it (line 59).

When node *C1* receives an OptimizationPropose message, containing a new proposed parent (line 65), it verifies that the message was sent by its current parent, discarding it if it is not. After this, it sends an OptimizationProposeRequest message containing itself and the *proposer* to the *C2* node, indicating it wishes to become its child. When the proposed parent receives this message (line 69), it verifies that the *proposer* node is still its parent and that node *C1* is also its sibling, if yes, then it adds the node as its new child, and replies with an *OptimizationProposeRequestReply*. This message contains a boolean flag, signalling if the node was added as a child or not. When this message is received (line 76) by node *C1*, it also verifies that the *proposer* node is still its parent, aborting the process if it is not, and adds the proposed node as its parent.

After this process is complete, if not aborted, the *C2* node becomes the parent of node *C1*, and the *proposer* node has fewer children, reducing its group size towards the configured maximum (as the *proposer* node only executes this mechanism if its children

number exceeds the configured amount). Furthermore, when possible, node *C1* obtains a new node with lower latency than its current latency to the *proposer* node, effectively improving the overall overlay cost, while maintaining group sizes within bounds.

It is important to note that since the mechanism limits the latency downgrade for each new parent connection, it does not guarantee that group sizes are bounded. Although it would be possible to bound the number of nodes per group if this condition were ignored, then this mechanism would conflict with the third mechanism, described in Section 4.2.2.3

A final mechanism employed in the management of the active view maintenance that is omitted from the pseudocode is responsible for ensuring that groups sizes do not become too small. In summary, every node periodically verifies the number of peers that are its siblings, if this number is lower than a certain (configurable) bound, the node "rolls a dice" (essentially generates a random number and verifies if it is lower/above a certain configurable threshold) to decide if it should abandon the current group in favour of joining its grandparents'. If the dice roll is positive, then the node sends a message to its grandparent asking to become its child. When the grandparent receives this message, it verifies if the exchange causes a loop in the tree and adds the node to its children if it does not. Then, it notifies the sender of the message by sending a message reply containing a boolean value representing if the sender node was accepted as a child or not.

It is important to mention that the aforementioned threshold of the "dice roll" (and consequently the probability of the node remaining in the current group) changes as a function of the size of the group, decreasing proportionally to the difference between the configured minimum group size and the current nodes' group size.

4.2.2.3 Passive view maintenance & Oportunistic improvement

The third mechanism of the devised membership algorithm is called "Passive view maintenance & Oportunistic improvement", and as its name suggests, it is responsible, in each node, for creating and maintaining an auxiliary pool of nodes in the overlay which are not descendants of the local node. This pool serves two purposes: the first is to enable fault tolerance in the overlay without having to rely on the landmarks, the second is to enable the self-improvement of the overlay, through parent exchanges toward "closer" parents (according to latency values).

There are three components of the Node type (the ID, Coordinates and the version of each node) which were present in the pseudocode of the previous mechanisms, but their explanation was omitted given they are only relevant to the behaviour of the this mechanism. We now explain each in detail and how they are obtained:

1. The **ID** of each node is a collection of string segments, where each node's ID is the concatenation of every segment of every ascendant of the node with its own segment. Each node's segment is generated by each parent whenever a new node requests to be its child. An example of the resulting IDS can be observed in Figure 4.2, where,

for example, all nodes that are descendants of the node with ID "DC2", begin their ID with "DC2" (e.g. "DC2/G1"). In this case, the ID is made up of the segments: "DC2" and "G1". This hierarchical ID structure gives each node enough information to evaluate if any other node in the overlay is its descendant (by verifying if one of ID is contained in the other), therefore allowing nodes to evaluate if a change of parent in the overlay would cause a cycle in the tree. This ID structure also allows nodes to check the level of any node, as the number of segments of an ID is the same as the level of that node in the tree (where landmarks are the root of the tree).

2. The **coordinates** of each node is an array of integers, where each position contains the obtained latency toward the corresponding landmarks. These coordinates are used as a heuristic for measuring the distance to new nodes in the passive view (to be potential parents).

3. The **version number** of a node is a monotonic integer that is incremented at every ID change and child addition or removal. The versions number allow nodes to replace outdated entries in their passive views. For example, if a node switches parent, it also changes its ID (and increases its version number). In this situation, other nodes must update their passive views to reflect this change, this is important as it may prevent them from measuring the latency to a node that could potentially be an incompatible parent because it would cause a cycle in the tree previously (i.e., before the change of its ID).

With these concepts explained, we now present the pseudocode for the mechanism (Algorithm 3). Similar to previous algorithms, the first lines declare the necessary state maintained by each node to execute the mechanism, which is composed of a set of nodes materializing the passive view of the node (line 5). In the following lines, we may observe the mechanism for filling this set. This mechanism is a periodic procedure triggered at pre-configured intervals which causes the emission of a new random walk message (line 7), the created random walk message contains (1) a random sample of nodes from the emitting node's passive view and active view, (2) the original sender's ID, and (3) an integer representing the messages' time-to-live (TTL). This message is then sent to a node that is not a descendant of the sender.

Whenever this message is received (line 12), if it has travelled more than a certain (configurable) number of hops, then the receiving node removes a (also configurable) number of nodes from the sample. Conversely, if the message has not yet travelled the number of hops, the previous step is skipped. After this, the node merges the removed nodes (if there are any) into his passive view and adds a random sample of nodes from his own passive and active view to the sample. If the configured maximum sample size is exceeded, then a number equal to the number of inserted nodes is discarded from the sample at random (lines 16-25). The intuition behind skipping a certain number of hops before removing nodes from the sample is to promote nodes collecting information from

Algorithm 3 Membership protocol (Passive view maintenance)

```
1:  State
2:      parent : Node                                                          ▷ defined in join
3:      children : dict<string,Node>                                           ▷ defined in join
4:      siblings : dict<string,Node>                                           ▷ defined in join
5:      pView : set<Node>
6:
7:  Every config.RandWalkPeriodicity Do
8:      sample ⟵ getRandSample([pView + allNeighs + children + parent + siblings], config.NrPeersToMergeRandWalk)
9:      target ⟵ getRand(parent + siblings)
10:     sendMessage(RandomWalk<sample + self, config.RandWalkTTL, self.ID, self.IP>, target)
11:
12: Upon receive( RandomWalk<sample, ttl, nID, orig>, sender) Do
13:     nrNodesToRemove ⟵ config.NrPeersToMergeRandWalk
14:     if config.RandWalkTTL - ttl < config.NrStepsToIgnore then:
15:         nrNodesToRemove ⟵ 0
16:     updateNodesToHigherVersion(sample, pView)
17:     ascNeighs ⟵ set(parent + siblings)
18:     allNeighs ⟵ set(ascNeighs + children)
19:     toAdd ⟵ getRandSample(excludeDescendantsOf(pView + allNeighs / sample,self.ID), config.NrPeersToMergeRandWalk)
20:     toRemoveFromSample ⟵ getRandSample(sample, nrNodesToRemove)
21:     sample ⟵ sample.removeAll(toRemoveFromSample)
22:     pView ⟵ excludeDescendantsOf(toRemoveFromSample + pView, self.ID)
23:     pView ⟵ pView.removeAll(allNeighs)
24:     pView ⟵ trimSetToSize(pView, config.MaxEViewSize)
25:     sample ⟵ trimSetToSize(sample + toAdd + self, config.config.MaxRndWalkSampleSize)
26:     target ⟵ getRand(excludeDescendantsOf(allNeighs, nID)
27:     if target == nil || ttl == 0 then
28:         sendMessageSideChannel(RandomWalkReply<sample>, orig)
29:     else
30:         sendMessage(RandomWalk<sample, ttl-1, nID, orig>, target)
31:
32: Upon receive(RandomWalkReply<sample>, sender) Do:
33:     sample ⟵ excludeDescendantsOf(sample, self.ID)
34:     updateNodesToHigherVersion(sample, pView)
35:     sample ⟵ excludeNodesInActiveView(sample)
36:     pView ⟵ trimSetToSize(pView + sample, config.MaxEViewSize)
37:
38: Every config.OportunisticOptimizationTimeout Do
39:     toMeasureRand ⟵ getRandSample(pView, len(pView)) // shuffle sample
40:     toMeasureBiased ⟵ sortByEuclideanDist(pView / toMeasureRand)
41:     measuredNr ⟵ 0
42:     for i=0; i < len(toMeasureRand) && measuredNr < config.ToMeasureRand ; i++ do
43:         if canBecomeChildrenOf(p) then
44:             measuredNr++
45:             measurePeer(p)
46:     measuredNr ⟵ 0
47:     for i=0; i < len(toMeasureRand) && measuredNr < config.toMeasureBiased ; i++ do
48:         if canBecomeChildrenOf(p) then
49:             measuredNr++
50:             measurePeer(p)
51:
52: Upon peerMeasured(p, measuredLatency) Do
53:     latencyImprovement := parent.currentLatency() - measuredLatency
54:     if latencyImprovement >= config.MinLatencyForImprovement then
55:         sendMessageSideChannel(OportunisticImprovementReq<self>,p)
56:
57: Upon receive(OportunisticImprovementReq<p>,sender) Do
58:     if isDescendant(p.ID,self) then
59:         sendMessageSideChannel(OportunisticImprovementReqReply<false>,sender)
60:     else
61:         addChildren(sender)
62:         sendMessageSideChannel(OportunisticImprovementReqReply<true>,sender)
63:
64: Upon receive(OportunisticImprovementReqReply<answer>,sender) Do
65:     if answer then
66:         disconnectFromCurrentParent(parent)
67:         addParent(sender)
68:
69: Procedure canBecomeChildrenOf(c, parent)
70:     if (c.nrChildren > 0 && parent.ID.level() >= c.ID.level()) then
71:         return false
72:     return parent.nrChildren > 0 && !isDescendantOf(parent.ID, c) && !isDescendantOf(c, parent.ID)
73:
74: Procedure isDescendantOf(nodeID, PotentialDescID)
75:     return PotentialDescID.Contains(nodeID)
76:
```

nodes further away in number of hops from the source of the random walk. After this, the message TTL is decreased by one and its value is evaluated: if the TTL of the message is higher than 0, then the node forwards the message to a random node from its active view that is not a descendant of the original sender (descendants are excluded by verifying if their ID is not contained in the original senders'). If there is no valid target to forward the message to, then the node sends (via a temporary connection) a RandomWalkReply message to the original sender of the random walk with the sample (lines 27-30).

Whenever a node receives a RandomWalkReply it merges the received sample with its passive view, excluding all of its descendants and nodes in the active view (lines 32-36).

As the overlay evolves with time, the passive views of nodes fill with nodes that are not descendants of the node in question, meaning they are suitable for latency optimizations and fault recovery (in case a parent dies). The (periodic) procedure responsible for evaluating these nodes can be observed in lines 38, where the node selects two (configurable) samples from the passive view to evaluate: a random sample and a sample based on the euclidean distance of the coordinate pair. Each node selected for this sample (candidates) must satisfy the following conditions (lines 43 and 48):

1. Have more than one direct descendant (i.e. more than one child).

2. If the candidates' level (obtained from the ID) is lower than the measuring nodes' and the measuring node has more than 0 children, then the candidate is excluded. This prevents nodes with multiple children from going down in levels and instead favours nodes with no children "climbing" to higher levels of the tree.

After the measurements are issued, whenever a peerMeasured event is triggered (line 52), the node compares the current latency of its parent with the measured nodes' latency: if that latency is lower than the current parents' latency by a configurable threshold, then the measuring node will send an OportunisticImprovementReq message to the measured node. When the measured node receives this message (line 57), it checks that the receiving node is not a descendant of the sender (to prevent the creation of loops in the tree) and replies with an OportunisticImprovementReqReply message containing a boolean value representing whether the node was accepted as a child or not.

When this message is received, (line 64), if the exchange was accepted, the node disconnects from the parent node (via a special message), establishes a connection to the new parent, and updates its own ID and version.

4.2.2.4 Fault tolerance

Fault tolerance in this protocol is triggered whenever a node detects its parent has failed. This can be achieved either by the PHI-accrual failure detector provided by the Node Watcher (Section 3.2) or by the failure of a TCP connection that triggers a notification to

the protocol. Whenever this occurs, the node first attempts to fall back to its grandparent (provided via the periodic information in Section 4.2.2.2), if this fails, it falls back to any node in its passive view that is not a descendant.

When a node falls back to another node, it sends a FaultRecovery message containing its ID and sets up a timeout timer for each fault recovery attempt. Nodes that do not reply to FaultRecovery messages within the specified timeout are considered to be failed and are removed from the passive view. If this view becomes empty, then the node starts the join mechanism again (Section 4.2.2.1).

4.2.3 Summary

In this section, we provided a detailed explanation of the behaviour of the membership protocol. We began by explaining how nodes join the network using a greedy depth-first search to find a suitable low-latency node in the network with more than zero children. Then, after this low-latency parent is established, we described how information between nodes and their parents is exchanged over time and how the parent node coordinates with its children in an attempt to maintain the group size within a certain bound and (when possible) reduce the systems' latency.

Lastly, we explained how nodes obtain information about other random nodes in the network and how that information is employed to perform both latency optimizations that reduce the total overlay network latency. Lastly, we covered fault recovery in the protocol, which occurs whenever the parent of a certain node crashes.

4.3 Aggregation protocol

Provided with a membership protocol capable of coordinating nodes into building an efficient tree structure, we now discuss how we leveraged it to provide efficient abstractions for performing aggregation/collection of metrics about the execution of nodes (or services) executing in the system in a decentralized manner. In this section, we cover the three implemented aggregation primitives: (1) tree aggregation, (2) neighbourhood aggregation, and (3) global aggregation, starting with tree aggregation.

4.3.1 Tree aggregation

Tree aggregation is a mechanism that embeds an **aggregation tree** into the overlay protocols' to collect an aggregated value for all nodes which are descendants of the node performing this mechanism (also denoted the **root of the aggregation tree**). This mechanism can be performed by any node in the system, and if two different nodes are aggregating the same values and a node is a descendant of the other, then the descendant node will (when possible) reuse the values of the already existing aggregation tree by embedding its tree into the ascendants'.

The pseudocode for this mechanism can is provided in Algorithm 4. The first lines define the necessary state to execute this aggregation mechanism (line 1), starting by the active view, composed by: the parent, children, and siblings of the node (maintained by the overlay protocol with changes to it propagated through notifications). In addition, the state also contains three maps, the first map, called *tIds*, contains the necessary metadata for each aggregation tree. Each value of this map is composed of: (1) the height of the tree, (2) the merge function, (3) the query to generate local values, (4) the periodicity to export values, (5) the output metric name, (6) the ID of the corresponding timer, (6) a boolean value representing if the value should be exported locally, (7) a boolean representing if the parent is also in the tree (and the node must propagate values to it or not), and finally, (8) the ID of the tree from the parent's perspective (or nil, if the node has no parent). The second map (denominated *lastSeen*) contains a timestamp for each tree, representing the last time the parent has sent a message refreshing the existence for that tree. Finally, the *childValues* map contains, for each tree, the values emitted by the children and the timestamp of their reception.

Nodes begin executing this mechanism whenever the API sends a StartTreeAggregationRequest request to the protocol, which contains the maximum height of the tree, the merge function, the query to obtain the local value, the periodicity at which to execute the aggregation procedure, and the resulting metric name. Upon the reception of this request (line 9), the node creates the ID for the aggregation tree by hashing the concatenation of: (1) the tree height, (2) the merge function, (3) the query, (4) the mechanism periodicity, and (5) the resulting metric name. By using this combination of parameters for the hash, nodes guarantee that aggregation trees with the same height (from the hashing nodes' perspective) have the same ID. Whenever two trees have the same ID, the node that detects this may reuse the already existing trees' aggregation results, thus not increasing the necessary messages to collect the metric values.

After this, the node adds the tree ID to its local aggregation *tIds* map, first checking if there was already an existing tree with the same ID (meaning it may reuse the existing tree for obtaining the requested values). If there is, then it sets a flag signalling it should also save the values for that tree locally. Otherwise, it adds a new entry to the *tIds* map and sets up a periodic timer for that aggregation tree (lines 10-16).

In order to federate nodes into their aggregation trees, nodes periodically (using configured intervals) broadcast to their children a Subscription message containing the metadata (including the ID) of the aggregation trees they are the root of (line 39). Whenever this message is received (line 48), for each received ID, if it was previously present in the *tIds* map, the child marks the parent as a subscriber to that tree, and refreshes the timestamp associated with the tree in the *LastSeen* map. Conversely, for each received aggregation tree that was not previously in the *tIds* map, it sets a new periodic timer called ExportTreeAggTimer, and adds the ID to the *tIds* map along with the tree metadata. Lastly, the node subtracts by one each of the received tree TTLs and sends a Subscription containing the ones with TTL higher than zero (or equal to -1) to its

Algorithm 4 Tree aggregation

```
1:  State
2:      parent : Node                                              ▷ Defined by the overlay protocol
3:      children : dict<string,Node>                               ▷ Defined by the overlay protocol
4:      siblings : dict<string,Node>                               ▷ Defined by the overlay protocol
5:      tIds ⟵ map()
6:      lastSeen ⟵ dict<string,>
7:      childValues : dict<string,dict<string,<value, timeStamp»> ⟵ dict()
8:
9:  Upon StartTreeAggregationRequest(tHeight, mergeF, query, periodicity ,outmName) Do
10:     tId ⟵ hash(tHeight + mergeF + query + periodicity + outmName)
11:     if tId in tIds then
12:         <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
13:         tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, true, isParentSub, ptId>
14:     else:
15:         timerId ⟵ registerPeriodicTimer(ExportTreeAggTimer(tId), periodicity)
16:         tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, true, false, nil>
17:
18: Upon ExportTreeAggTimer(tId) Do
19:     <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
20:     if isParentSub && timeSince(lastSeen[tId]) > config.treeAggExpiration then
21:         if !isLocal then
22:             tIds.delete(tId)
23:             lastSeen.delete(tId)
24:             cancelTimer(timerId)
25:             return
26:     removeOldChildrenValues(childValues[tId])
27:     res ⟵ aggregateValues(mergeF, resolveQuery(query), childValues[tId])
28:     if isLocal then
29:         storeLocalVal(res, outmName)
30:     if isParentSub then
31:         sendMessage(PropagateTAggValues<ptId, res>, parent)
32:
33: Upon receive(PropagateTAggValues<tId, res>, sender) Do
34:     if tId in tIds and sender in children then
35:         if tId not in childValues then
36:             childValues[tId] = map()
37:         childValues[tId][sender] = res, time.Now()
38:
39: Every config.PropagateTAggTimeout seconds Do
40:     toSendArr ⟵ set
41:     for tId in tIds do
42:         <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
43:         if isLocal then
44:             toSendArr.append(<max(tHeight -1, -1), mergeF, query, periodicity ,outmName, tId>)
45:     for c in chilren do
46:         sendMessage(RefreshTreeAggFunc<toSendArr>, c)
47:
48: Upon receive(RefreshTreeAggFunc<tAggs>, sender) Do
49:     if parent == sender then
50:         toSendArr ⟵ set
51:         for <tHeight, mergeF, query, periodicity, outmName, ptId> in tAggs do
52:             tId ⟵ hash(tHeight + mergeF + query + periodicity + outmName)
53:             if id in tIds then
54:                 <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
55:                 lastSeen[id] ⟵ time.Now()
56:                 tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, true, ptId>
57:                 if !isLocal && <max(tHeight -1, -1) == -1 || <max(tHeight -1, -1) > 0 then
58:                     toSendArr.append(<max(tHeight -1, -1), mergeF, query, periodicity ,outmName, timerId, tId>)
59:             else
60:                 toSendArr.append(<max(tHeight -1, -1), mergeF, query, periodicity ,outmName, timerId, tId>)
61:                 tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, false, true, ptId>
62:                 registerPeriodicTimer(HandleTreeAggTimer(tId), periodicity)
63:         for c in chilren do
64:             sendMessage(RefreshTreeAggFunc<toSendArr>, c)
65:
```

children. This means that if a tree has a single root node, and that node crashes or stops propagating SUBSCRIPTION messages, all other nodes belonging to that tree will stop sending any more SUBSCRIPTION messages.

Whenever the aforementioned periodic timer called EXPORTTREEAGGTIMER triggers for a certain aggregation tree, (line 18), the node checks if it has expired (i.e. if the parent stopped refreshing the tree), if it has, and the node is not a root of that tree, then it cancels the timer and deletes any related metadata. Conversely, if the node is a root of the tree, it sets the flag representing whether the node should propagate to the parent as "false". If the tree has not expired, the node evaluates the trees' query (this procedure will be explained in further detail in Section 4.4), obtaining its local value and merging it (using the supplied aggregation function) with all the values sent by its children, producing the final aggregated result (before merging the values, the node excludes all values with a timestamp older than a configurable duration). Afterwards, if the node is a root of the tree, it stores the value locally, and if the flag signalling it should propagate to the parent has the value "true", it sends a PROPAGATETAGGVALUES message to the parent containing the obtained value. Upon the reception of this PROPAGATETAGGVALUES message by the parent, (line 33), it verifies if it has the corresponding aggregation tree in its local *tIds* map, discarding the message if it is not present, and, stores the received value into its *childValues* map.

Although it is omitted from the pseudocode, it is essential to mention that for nodes that are roots of the aggregation trees, it is also possible (according to a configuration parameter) to store the neighbours' values locally (without merging with the local or any other neighbours' values). We believe this feature is useful for resource management applications, for example, in the case of an application that needs to deploy service replicas according to geographical proximity to a certain target, this feature can be useful for obtaining a histogram of latitudes and longitudes for all nodes "behind" a certain node in the active view. Using this information, a resource management application can have a sense of which direction to send messages to in order to find nodes located in a given geographical region (by recursively sending messages to nodes in the active view that have frequencies closer to the desired geographical region).

4.3.2 Neighbourhood aggregation

Neighbourhood aggregation is the mechanism responsible for collecting metrics from neighbouring nodes. This feature is useful in resource management scenarios such as: whenever a node needs to perform service replication/migration, it may collect metrics related to the capacity and resource consumption of nearby nodes (in terms of hop distance) and evaluate which peer is the best candidate before performing such actions.

In essence, this mechanism behaves similarly to a hop-based Pub-Sub system. Although the designated name for this protocol is neighbourhood aggregation, the nodes actually do not perform aggregation of the values. Instead, nodes collect all the values

provided by their nodes within a (configurable) hop range. Similarly to tree aggregation (Section 4.3.1), a node performing this mechanism creates an aggregation tree rooted on itself by broadcasting SUBSCRIPTION messages periodically with a configurable hop-based range (or TTL) for all its neighbours. Nodes that receive this message become federated in the aggregation tree, decrease the message TTL, and if the TTL is more than zero, and rebroadcast it to peers in the active view in a way that creates an acyclical graph. Afterwards, for each tree, all federated nodes periodically propagate their locally obtained values (from evaluating the supplied query) towards the root of each tree using the reverse path established by the SUBSCRIPTION messages.

Trees have associated IDs, generated using hashing in a similar manner to the tree algorithm defined in Section 4.3.1 (without using the level in the hash process), and consequently, all nodes collecting the values from the same query using this mechanism will have trees with equal IDs. Nodes belonging to overlapping trees (i.e. in the hop range of two different nodes collecting values in this manner) only generate values periodically for one of the trees and propagate the generated value towards the direction of the multiple tree roots, preventing unnecessary query evaluations. In addition, nodes in overlapping trees, when possible, also deduplicate the SUBSCRIPTION messages (maintaining the federation of these trees).

This mechanism, similarly to Tree Aggregation (Section 4.3.1), is triggered via a request from the API containing, among other parameters, the query to obtain the local values, the hop range, and the target periodicity to collect the values. The receiver of the request, denoted the **root of the aggregation tree**,(illustrated by node A in Figure 4.3) creates the ID for that aggregation tree by hashing a combination of the metric name and the periodicity. This process makes nodes with equal parameters have equal IDs and become federated to the same tree. After the ID generation, the node begins propagating a SUBSCRIPTION message periodically to its immediate neighbours (illustrated in step 2 of Figure 4.3) containing the ID of the tree, the TTL, the query to obtain their local values, and the mechanism periodicity.

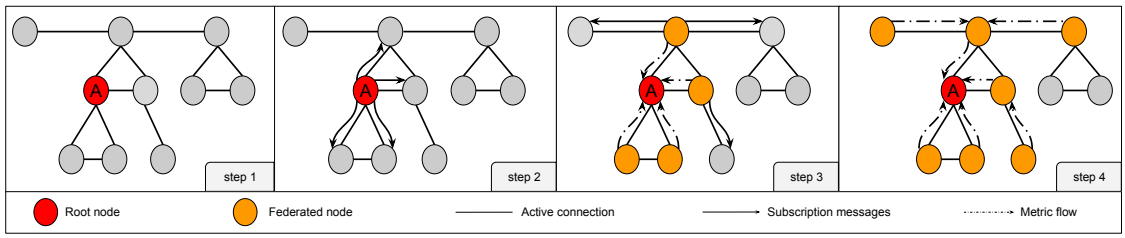

Figure 4.3: Neighbourhood aggregation subscription process (TTL=2)

Whenever a node receives this SUBSCRIPTION message, it performs the following steps:

1. Verifies the message came from a node contained in the active view, if it did not, the

message is discarded.

2. Stores, for that sender, the ID of the tree, the TTL of the message and a timestamp of the current time. If there is already such an entry, instead, its timestamp is refreshed.

3. Decreases the TTL of the message by one.

4. If the message TTL is 0, the node returns from this procedure.

5. Following, the node performs the following steps to decide where to broadcast to:

   a) If the message came from a parent or a sibling, the node broadcasts the message to its children.

   b) If it came from a child, then the node broadcasts the message to the parent and its siblings.

   c) Before broadcasting to any node, the sender verifies if it has sent a SUBSCRIPTION message with a higher or equal TTL than the TTL from the received one to these nodes in the last (configurable) time window. If it has, the sender skips the message emission for that node.

With this, in case two different nodes in the system are collecting the same metrics and using the same periodicity, the SUBSCRIPTION messages are not sent unnecessarily to nodes already subscribed to that tree. This process is illustrated by node *A* in Figure 4.4, where it receives the SUBSCRIPTION message and does not propagate it to its siblings nor parent, as it is already federated to the tree (rooted on itself with TTL= 2) and has sent a SUBSCRIPTION message to its siblings and parent in the previous step.

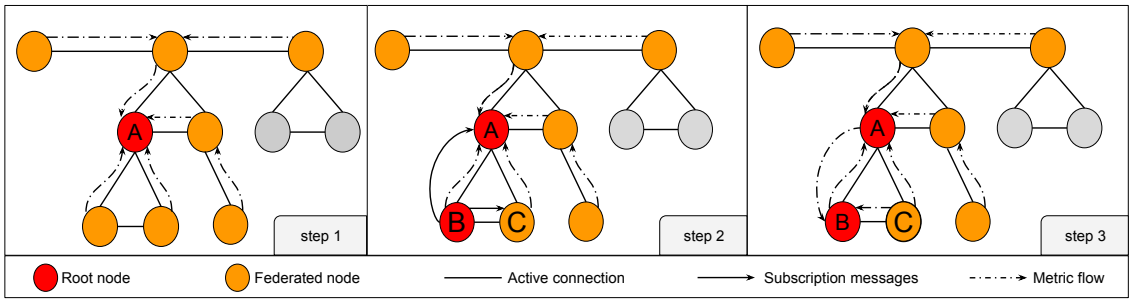

Figure 4.4: Neighbourhood aggregation second subscribe (TTL=2)

After nodes become federated in the trees, they begin to periodically evaluate the supplied query and obtain their local metric values (storing them locally if they are a root of that tree). When a node obtains its local metric value, it propagates a message containing the metric value and a hop counter to every other node in its active view that has sent a SUBSCRIPTION message within a configurable time frame with a TTL lower or

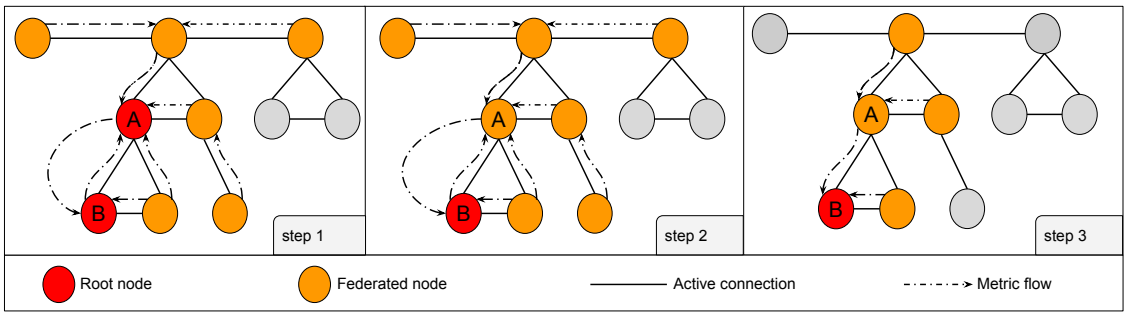

Figure 4.5: Neighbourhood unsubscribe (TTL=2)

equal than the messages'. Nodes that receive the propagation of these values increase the hop counter and repeat this process. This process is illustrated in Figures 4.3 4.4 and 4.5.

Lastly, nodes periodically verify, for each tree, the time passed since the reception of the last SUBSCRIPTION message. If it exceeds a configurable time frame, the entry is removed, and that node will stop receiving the propagation of metric values. When nodes remove the expired entry, if there is no other entry for that tree ID and the node is not a root of that tree, they delete all metadata related to that tree and stop propagating metric values (illustrated in Figure 4.5).

4.3.3 Global aggregation

Global aggregation is the mechanism executed whenever a certain client wishes to obtain a summarized global view of the system (e.g. the total number of nodes in the system). This process, similarly to the ones described in Sections 4.3.1 and 4.3.2, is started via a request from the API and functions by federating nodes of the system into **aggregation trees**, rooted on the nodes that are collecting the aggregate values. In global aggregation, all nodes participate in all aggregation trees, either as a root (if they wish to collect the globally aggregated value), or alternatively as aggregator nodes. Additionally, if a tree has multiple root nodes, then nodes, when possible, reuse the metric values from the first tree root towards the other roots, and deduplicate the maintenance mechanisms of the aggregator trees.

This mechanism is inspired in the work from Mirage protocol [11], which employs an aggregation technique that leverages a tree-shaped overlay to allow the computation of a globally aggregated value in a decentralized and efficient manner by every node in the system. This is achieved by having every node periodically broadcasting to every neighbouring peer their aggregated value minus the neighbours' contribution. Nodes that receive this broadcast merge all received contributions with the locally generated one. The continuous execution of this procedure results in all nodes obtaining the global aggregated value without resorting to aggregating the values toward a single node in the system.

In this mechanism, we leverage the same aggregation technique to collect globally aggregated values in multiple (but not necessarily all) nodes of the system in a decentralized manner. However, unlike the original Mirage protocol, instead of only performing aggregation of a single metric value, we generalize the approach to allow the on-demand creation and teardown of aggregation trees, rooted in one or more nodes, with all roots collecting the globally aggregated value in a decentralized manner. The relaxation of these constraints creates additional challenges regarding the transmission of redundant messages to maintain the multiple aggregation trees and the decommissioning of the trees, which we attempt to solve in this work. It is important to mention that, in a scenario where all nodes of the system are roots of the aggregation tree, this mechanism behaves similarly to the originally proposed in [11], with the only difference being of allowing the on-demand start and decommission of the aggregation process.

The state necessary for the execution of this mechanism (presented in Algorithm 5 lines 1 to 6) starts with the active view of the node executing the mechanism, composed by the parent, children and siblings of the node in question. Changes in this view are propagated by notifications emitted by the overlay protocol (these are omitted from the pseudocode for readability). In addition, the state contains a map denominated *lastTimeSent* that contains for each tree ID, and for each neighbouring node, a timestamp corresponding to the last time that node has refreshed the existence of the tree with that ID. Additionally, each node also maintains a *neighValues* map, which stores, for each tree, the propagated neighbour values and a timestamp of the reception of these values. Finally, each node owns a map denominated *tIds*, which holds the metadata needed to manage the aggregation trees, composed by: (1) a difference function, used to remove the contributions of a certain node from an aggregated value, (2) the merge function, used for merging two or more values into an aggregated value, (3) the query to obtain the local values, (4) the resulting output name for the aggregated metric, (5) the periodicity at which to collect the aggregated value, (6) a boolean representing if the node executing the protocol is a root of the aggregation tree, and finally, (7) a map called *aggNeighs* which contains the peers that are interested in receiving values for that tree (previously referred to as the aggregator nodes).

This mechanism is initiated with the reception of a request from the API (line 9), which contains multiple parameters: (1) the difference function, (2) the merge function, (3) the query to obtain the local value (4) the periodicity to perform this mechanism, and (5) the resulting metric name (to label the output values). Upon reception of this request, the node hashes the concatenation of the difference function, the merge function, the query, and the periodicity of the request, obtaining the tree ID, which will be common to every node in the tree. After the ID generation, the node checks if there already is a tree with that ID present in its local *tIds* map, setting as true the variable which denotes if the node should save the aggregated value locally. If there is no tree with such ID previously present, the node sets up a new ExportGlobalAggTimer with the provided periodicity and creates a new entry in the *neighValues* map for that tree.

Algorithm 5 Global aggregation

```
1:  State
2:      parent : Node                                                          ▷ Defined by the overlay protocol
3:      children : dict<string,Node>                                           ▷ Defined by the overlay protocol
4:      siblings : dict<string,Node>                                           ▷ Defined by the overlay protocol
5:      lastTimeSent : dict<string, dict<string, timeStamp» ⟵ dict()
6:      neighValues : dict<string, dict<value, timeStamp» ⟵ dict()
7:      tIds : dict<string, <string, string, timeDuration, string, string, boolean, dict< string, timeStamp» ⟵ dict()
8:
9:  Upon StartGlobalAggregationRequest(diffF, mergeF, query, periodicity ,outmName) Do
10:     tId ⟵ hash(diffF + mergeF + query + periodicity)
11:     if tId in tIds then
12:         <diffF, mergeF, query, periodicity, outmName, timerId, isLocal, aggNeighs> ⟵ tIds[tId]
13:         tIds[tId] ⟵ <mergeF, query, periodicity, outmName, timerId, true, aggNeighs>
14:     else
15:         timerID ⟵ registerPeriodicTimer(ExportGlobalAggTimer(tId), periodicity)
16:         tIds[tId] ⟵ <mergeF, query, periodicity, outmName, timerId, true, dict()>
17:         neighValues[tId] = dict()
18:
19:
20: Every config.PropagateGAggTimeout seconds Do
21:     toSendArr ⟵ set
22:     for tId in tIds do
23:         <diffF, mergeF, query, periodicity, outmName, timerId, isLocal, aggNeighs> ⟵ tIds[tId]
24:         for <node, timestamp> in aggNeighs do
25:             if timeSince(timestamp) > config.SubExpirationDuration then
26:                 aggNeighs.remove(node)
27:         if aggNeighs.length == 0 && !isLocal then
28:             tIds.remove(tId)
29:             continue
30:         if isLocal then
31:             toSendArr ⟵ toSendArr + <diffF, mergeF, query, periodicity, outmName, tId>
32:     PropagateGAggTrees(toSendArr, parent + children)
33:
34: Upon receive(RefreshGaggTree<gAggs>, sender) Do
35:     gAggTreeArr ⟵ set
36:     for <diffF, mergeF, query, periodicity, outmName, tId> in gAggs do
37:         if id in tIds then
38:             gAggTreeArr.append(<diffF, mergeF, query, periodicity ,outmName, timerId, tId>)
39:             neighValues[tId] = dict()
40:             tIds[tId] ⟵ <diffF, mergeF, query, periodicity ,outmName, timerId, false, <sender: time.Now()»
41:             registerPeriodicTimer(HandleTreeAggTimer(tId), periodicity)
42:         else
43:             <diffF, mergeF, query, periodicity ,outmName, timerId, isLocal, aggNeighs> ⟵ tIds[tId]
44:             aggNeighs[sender] ⟵ time.Now()
45:             tIds[tId] ⟵ <diffF, mergeF, query, periodicity ,outmName, timerId, isLocal, aggNeighs>
46:             if isLocal then
47:                 continue
48:     if sender == parent then
49:         PropagateGAggTrees(gAggTreeArr, children)
50:     if sender in children then
51:         PropagateGAggTrees(gAggTreeArr, children - sender + parent)
52:
53: Upon ExportGlobalAggTimer(tId) Do
54:     <diffF, mergeF, query, periodicity, outmName, timerId, isLocal, aggNeighs> ⟵ tIds[tId]
55:     removeOldNeighValues(neighValues[tId])
56:     localVal ⟵ resolveQuery(query)
57:     res ⟵ evalFunc(mergeF, localVal, neighValues[tId])
58:     if isLocal then
59:         storeValLocally(res, outmName)
60:     for <node, timestamp> in aggNeighs do
61:         sendMessage(PropagateGAggValues<tId, evalFunc(diffF, res, neighValues[tId][node]>, node)
62:
63: Upon receive(PropagateGAggValues<tId, res>, sender) Do
64:     if tId in tIds and sender in children || sender == parent then
65:         neighValues[tId][sender] = res, time.Now()
66:
67: Procedure UnknownPropagateGAggTrees(gAggTreeArr, nodeList)
68:     for node in nodeList do
69:         toSendToNode ⟵ set()
70:         for <diffF, mergeF, query, periodicity ,outmName, timerId, tId> in gAggTreeArr do
71:             if lastTimeSent[node][tId] == nil || time.Since(lastTimeSent[node][tId]) > config.RefreshMessageBackoff then
72:                 toSendToNode ⟵ toSendToNode + <diffF, mergeF, query, periodicity ,outmName, timerId, tId>
73:         lastTimeSent[node][tId] ⟵ time.Now()
74:         sendMessage(RefreshGaggTree<toSendToNode>,node)
75:
```

As previously mentioned, global aggregation allows the on-demand creation and decommission of aggregation trees. This process is defined in alg. 5 lines 21 to 32), where, as previously mentioned, nodes periodically send messages named RefreshGaggTree containing the aggregation trees they are the roots of to their children and parent and clear all entries in the *aggNeighs* that are older than a configured time frame. If a tree has no more entries in this map, and the *isLocal* flag is not set to true, then that tree is decommissioned.

Whenever the RefreshGaggTree message is received (Algorithm 5 line 34), the receiver adds the previously unknown trees into its local *tIds* map and sets up a periodic ExportGlobalAggTimer for each added tree (lines 37 to 42). Alternatively, if the tree was previously in the *tIds* map, the node refreshes the sender's entry in the *aggNeighs* map. Finally, the node removes the trees present in the message where it is also a root of (deduplicating the tree maintenance mechanisms) and forwards the remaining trees to every node in its active view, excluding the sender. Before transmitting the trees to each node, the node checks, for each tree, if it has transmitted a RefreshGaggTree message containing the same tree in the last (configurable) time frame. If it has, then it does not propagate that tree to that node. These verifications are performed to prevent trees from being refreshed multiple times unnecessarily.

4.3.3.1 Metric propagation

With the aggregation tree established, we now explain how the values are propagated and aggregated by each tree in the system. As previously mentioned, nodes set up an ExportGlobalAggTimer for each registered tree, whenever this timer triggers (alg. 5 line 53), the node first removes all out-of-date neighbour values for the corresponding tree (according to a configurable timeout) and evaluates the query, obtaining its local value. Then, using the neighbour values and the locally obtained value, it applies the merge function and obtains the globally aggregated value, which it stores locally if configured by the *isLocal* flag (lines 54 to 59). Then, for each entry previously in the *aggNeighs* map, it sends a PropagateGAggValues message with the aggregated value minus the node's contribution and the tree ID. (lines 60 to 61).

Finally, whenever nodes receive the PropagateGAggValues message (Algorithm 5 line 53) containing the aggregated value and the tree ID, they verify that it was sent from either the parent or the children and that the tree ID is in their local *tIds* map, discarding the message any one of these conditions is observed. Finally, they store the propagated value locally in their *neighValues* map for later use in computing the aggregated value.

In sum, nodes that are roots of their aggregation trees will, over time, receive aggregated values from their nodes, which are essentially sent and aggregated by all other nodes using the reverse path taken by theRefreshGaggTree messages. Given the fact that nodes only use their parents and children of the tree topology to forward messages (thus ensuring the tree has no cycles, as there is only a single path from any node to each

other node in the system), by propagating to a neighbour the resulting aggregated value without the effects of its contribution [11], multiple nodes in the system can simultaneously obtain the aggregated value efficiently and in a decentralized manner.

4.3.4 Summary

In this section, we presented the devised aggregation protocol. This protocol leverages the devised overlay protocol's tree structure to perform efficient propagation/aggregation of information in a decentralized manner. This protocol is coalesced by three decentralized information aggregation/collection primitives, which we believe to be useful for gathering partial or complete system information to perform decentralized resource management actions.

The first primitive is **tree aggregation**, where nodes, when requested, form aggregation trees (with configurable range) rooted upon themselves. These trees extend only to their descendants in the original overlay protocol tree, and nodes federated in these trees periodically merge their local value with their childrens' and send a message containing it to their parents. In case one descendant is executing the same primitive with a tree with the same range (from the descendants' perspective), it simply reuses the parent's tree to obtain the intended aggregated value.

The second primitive is **neighbourhood aggregation**, where nodes collect, on-demand and in a decentralized manner, the metrics of nodes in a hop-defined range. This mechanism behaves similarly to a pub-sub system, where nodes periodically propagate messages which federate other nodes in trees rooted upon themselves. Nodes in these trees propagate their local values periodically using the reverse paths taken by the federation messages. In this primitive, nodes (when possible) deduplicate federation messages and multiplex metric propagations.

Lastly, the third primitive called **global aggregation** is a primitive where nodes collect and aggregate, also on-demand and in a decentralized manner, a value that corresponds to the globally aggregated value of the system. This primitive is inspired by work from the state-of-the-art, however, it relaxes constraints imposed by the original work, such as performing the mechanism with only a partial set of the nodes being tree roots, in addition to allowing the technique to be performed in an on-demand fashion (based on API requests).

We believe these primitives are useful for resource management decisions such as, for example, maintaining a proportion of replicas to nodes, by employing **tree aggregation** collecting all the descendants' number of replicas and number of nodes, the tree roots can, in a decentralized and independent manner, perform replication or decommission of replicas to maintain the target value. The same applies to **global aggregation**, which allows nodes to, for example, collect the total number of replicas in the system and perform replication actions if they reach a lower than configured number. Lastly, **neighbourhood aggregation** allows nodes to collect information about nearby nodes, which can also be

used to improve system QOS by, for example, deploying a server closer to a client in terms of geographical distance.

It is important to mention that while in this work we present the protocol leveraging the overlay protocol defined in 4.2, the protocol is agnostic to which overlay protocol is executing underneath it, as long as it has the following characteristics: (1) forms one or more tree-shaped networks, whose roots are interconnected; (2) Nodes in these trees must be connected to their parents, children and siblings (also denoted by their active view) with bidirectional connections; and finally (3) the overlay protocol must also provide events for each node that is added or removed from the protocols' active view.

As previously mentioned, these decentralized aggregation primitives make use of both queries and aggregation functions to obtain the data and to summarize it (respectively). In this next section, we cover the monitoring module that, along with other features, interprets and processes the aforementioned queries and aggregation functions.

4.4 Monitoring module

The **monitoring module** is tasked with storing metrics, resolving queries regarding stored metrics, removing expired metrics, periodically evaluating registered alarms, triggering callbacks which the API then propagates to the client, among other features which we will now cover.

We believe it is, however, important to remember that the focus of this work is to provide a usable proof-of-concept of a decentralized monitoring framework, targeted for decentralized resource management solutions. Consequently, the focus of this module is to provide just enough functionality for a proof of concept, namely in aspects such as the storage of metrics in disk or the efficiency of the query language, which are interesting research challenges by themselves, however are orthogonal to the conducted work.

This module is composed by three different components (illustrated by fig. 4.6): the **query engine**, the **time-series database** (TSDB)), and the **alert manager**, whose roles in the system we now briefly explain:

1. The **time-series database** allows the insertion and retrieval of time-series data from the framework. This time-series database employs an in-memory index to retrieve one or more time-series according to a set of parameters.

2. The **query engine** is the component tasked with resolving queries made to the time-series database. This component maintains a set of sandboxes that evaluate user-provisioned queries that both extract data from the database and apply aggregation functions to it.

3. Finally, the **alert manager** manages the alarms issued by the API. In sum, this component periodically verifies the issued alarms' query using the **query engine** and propagates an event to the client whenever the condition is verified.

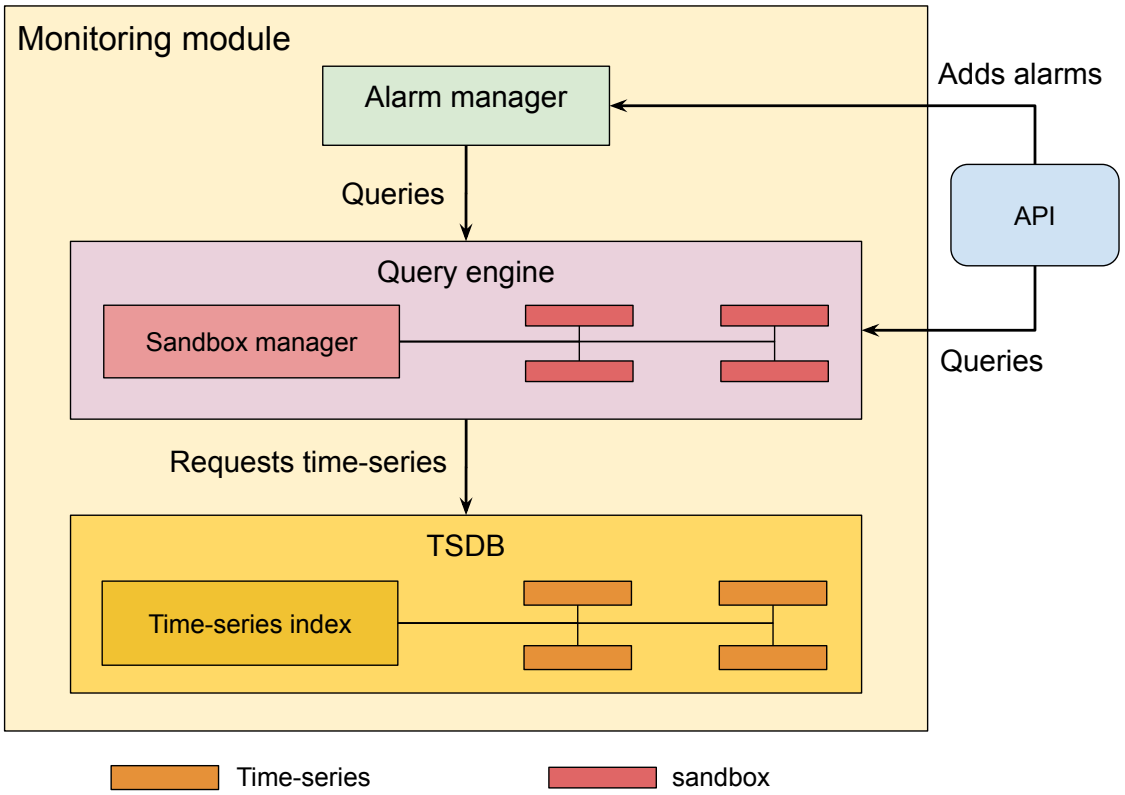

Figure 4.6: An overview of the monitoring module

In order to ease the explanation of these components, we first detail the structure of the metrics used in this framework, which has a similar structure to the metric types provided by InfluxDB [26].

4.4.0.1 Metric structure

In DeMMon, a metric is composed of four elements: first, the **name**, which is a string denoting the name of the stored information, the name should be a human-readable name which is self-describing (e.g. "CPU-USAGE"). The second element are the metric **tags**, which are a set of string pairs denoting attributes related to the metric that is stored, (e.g. the hostname or cluster name of the node that emitted it), next, we have the **value**, which contains the data associated with the observed metric, and finally, we have the **timestamp**, that contains the time at which the observation was taken. A typical example of a metric in the devised framework would be: name: "CPU-Usage"; tags: <host:nodeX>, value: 0.3, timestamp: "1609960731".

We believe it is important to mention that in order to remain as flexible as possible, the metric values do not have a defined type. In this system, clients may use custom types (as long as they are serializable using the JSON package provided by the Golang [18] package). This allows the devised framework to represent a multitude of different

information types, such as histograms, strings, string maps, among others, which offers a higher degree of flexibility for using this framework. For example, a decentralized service management system aimed at deploying service replicas in close proximity to the clients may use, for example, a histogram with pre-determined geographical classes. This way, this system would have a data structure that would ease finding a node in the desired geographical area.

Provided with the metric structure, we now explain how these are stored in the system.

4.4.0.2 Time-series database

In DeMMon, time-series are sequences taken at successive equally spaced points in time. In this system, time-series are stored only in memory and are indexed as a function of their name and tags.

In order for a certain metric (composed by: name, tags, and periodicity) to be inserted into the database, a **bucket** must first be created. A bucket is essentially a component that holds all time-series data with a certain name, periodicity and capacity. The periodicity denotes the interval at which the sequences of points are spaced (in time) and the capacity denotes the number of points stored in each sequence. For example, a time-series with a 5-second periodicity and a capacity of 12 holds all points from the last minute. Limiting the number of points per series allows the system to pre-allocate the memory necessary (using an array) for each time-series at the time of its creation.

Within a bucket, metrics are stored in a map and indexed by their tags. This is done by generating keys that are equal for each similar tag set, independent of its order: whenever a metric is inserted, the tag pairs are sorted alphabetically (by their key) and concatenated into a single string, producing the resulting metric key. Then, using the metric key, the metric value is inserted into the corresponding time-series (a new time-series is created for that tag set if there was none previously in the system).

Time-series advance time in an on-demand manner, meaning that, before returning values for any read or write request from a time-series, the system first verifies if its' oldest value has a timestamp outside of the time-series window (as time has passed since the last check). If it has, the system iterates the time-series' points from its oldest to the newest point and removes all points outside its time window. In order to remove unused time-series from the system, and decrease DeMMons' memory footprint, the time-series database component also periodically advances the time-series in time and removes any that become empty.

Concurrency is maintained using locking mechanisms, where operations that do not affect the state of the time-series are executed concurrently, and operations that would otherwise change the time-series status are executed sequentially.

4.4.0.3 Query engine

The query engine is a sub-component of the monitoring module, and it is responsible for evaluating the supplied text-based queries, transforming them into sets of instructions, and determining the final query result by executing the instructions. Keeping in mind the fact that the focus of this work is not the performance of the metric storage or the query language and that it is still a focal point of this work to be as flexible as possible in the query language, we opted for using javascript-based sandboxes to perform this work. This means that user-provided queries are essentially javascript code, and consequently, users have infinite control over the behaviour of their queries, provided these do not exceed the query timeout.

In order to provide this functionality, we opted for using the package Otto [50]. This package provides access to javascript "virtual machines" that essentially parse a string containing javascript code, and produce an AST from the parsed code. The produced ASTs are then executed and their result is returned by the VM. In order to allow users to access the time-series stored in the TSDB , the query engine provides every Otto virtual machine access to the following functions, which return time-series from the database:

1. Select(Bucket_Name, <Tag_set_regex), this function returns the time-series that are in the supplied bucket matching the provided tag set regex. The way the tag set regex matching works is: for every time series present in the specified bucket, if all of the tag keys in the supplied regex match all of the time series tags, then the time series is returned. An example of the usage of this function would be, for example: "select(CPU_USAGE, <host:.*, cluster:cluster1>)"

2. SelectLast(Bucket_Name, <Tag_set_regex>), this function behaves similarly to Select, however, it only returns the last inserted point in all matched time series.

3. SelectRange(Bucket_Name, <Tag_set_regex>, startDate, endDate), this function behaves similarly to "Select" and "SelectLast", however, it allows users to only extract a certain time-window from the matching time-series. To do so, it takes an additional argument, consisting of a time range, used to filter the points to return to the client.

We believe these functions cover the most common use cases for metric selection. These metrics, upon selection, can then be aggregated in any way the user specifies in the query (since they are composed of user-defined code). In order to ease the design of queries and prevent developers from rewriting the same aggregation functions, the query engine also provides some aggregation primitives which can be applied to one or more time-series such as: Max, Min and Average.

After the selection and aggregation of metrics, the resulting values are returned by placing them in a variable denoted "result" (in the user-defined code). Any query executed in DeMMon can only result in one of two types: a single time series or an array of

time series (following an interface defined by DeMMon). Given this, in order to allow the creation of new time series that follow this interface during the query process, there are two additional functions supplied to the virtual machines: the first is called "NewTimeSeries", which creates a new time series, this function takes as arguments the name, tags and values which will integrate the time series; second, we have the function called "NewObservable" which takes a value of any type and a timestamp, and creates a new metric point which can be added to time series.

With this, we now provide some examples of possible queries along with a brief description of what they do:

1. "Avg(SelectLast(CPU_USAGE, <host:.*, cluster:cluster1>))" this query selects the metrics with the name "CPU_USAGE" for all hosts which belong to cluster with name "cluster1" and returns the average of all the points.

2. "SelectLast(Nr_services, <tenant:tenant10>, startDate, endDate)" this query returns the timeseries for the metric called "Nr_services" for the tenant with name "tentant10" during the provided time range.

3. "SelectLast(Nr_replicas, <tenant:tenant10,service:service10>)" this query returns the timeseries for the metric called "Nr_replicas" for the tenant with name "tentant10" and service named "service10" during the provided time range.

With this, clients are able to obtain and manipulate data from the time series database using text-based queries. Furthermore, as the type of the value of each metric is not enforced, clients may store their metrics in custom data structures, tailored for their specific use-cases.

4.4.0.4 Alarm manager

The alert manager is the last component of the monitoring module, it is responsible for managing the **alarms** issued to the monitoring module. Alarms are essentially sets of parameters that contain, among other parameters, a condition to observe (e.g. the percentage of CPU usage) and a periodicity to observe this condition. Alarms are paramount to prevent applications from having to periodically query DeMMon to verify the condition themselves, effectively saving bandwidth. This component is essentially responsible for periodically verifying these alarms and issuing notifications to the client whenever their conditions are verified.

In DeMMon, an alarm contains the following parameters:

1. **Condition**. This is essentially a query (explained in 4.4.0.3) that must return a boolean value.

2. **Periodicity**. The periodicity denotes how often the condition is evaluated, and how often notifications are sent to the client that issued the alarm

3. **Backoff time**. The backoff time is a time duration that bounds the rate at which the monitoring module emits notifications, which would otherwise happen at the alarm periodicity every time the alarm is verified (e.g. if the supplied periodicity is low).

4. **Watch list**. The watch list is a set that, for every item, contains both a name and a set of tag filters. Whenever the alarm manager receives an alarm containing a watch list, in addition to performing the verification at the specified periodicity, it also performs the verification whenever any time series matching the watch list is changed. The rates at which the alarm is verified in this manner also respects the backoff time.

5. **CheckPeriodic**. CheckPeriodic is a boolean variable denoting if the alarm should be verified periodically. When false, the alarm manager does not check the metrics at every **"periodicity"** seconds (defined previously), effectively saving CPU time. This option is meant to be used together with the watch list, for example, for checking a parameter that is rarely altered.

The monitoring module, whenever it receives a new alarm, essentially adds it to a priority queue containing all the alarms. This priority queue uses the time of reception of the alarms plus their periodicity as their key to the queue. With this, alarms are sorted by the time at which they need to be verified. The monitoring module continuously obtains and removes the first item of the queue, containing the next alarm to verify out of all issued alarms and waits until it is time of verification of that alarm (i.e. the time of reception of the alarm plus its periodicity). Ater this, it evaluates the condition (emitting a notification to the client if the condition is verified), and re-adds the alarm to the queue with a key corresponding to the current time plus the alarms' periodicity. Whenever an alarm is verified, and the result of its condition returns "true", the alarm manager first verifies if it has emitted a notification for that alarm in the last "Backoff time" duration, and issues a notification to the client if it has not.

4.5 API

The API is the last module of the devised framework. As previously mentioned at the beginning of this chapter, the purpose of this module is to expose the functionality of the remaining components of the framework by mediating the interactions between the clients and the remaining modules via well-defined operations. In this section, we provide a brief overview of the API implementation and summarie its most relevant operations.

4.5.1 Overview

This API is coalesced by a message-based protocol performed via WebSockets [49]. The choice of using a message-based protocol is motivated by the fact that, contrary to traditional HTTP APIs, messages enable clients to receive events sent by the DeMMon servers (without requiring an explicit request). This feature is essential for both alerting (as triggered alarms need to be propagated to their issuing clients) and for issuing events such as active view changes (from the overlay protocol, defined in Section 4.2) to subscribed clients. Consequently, in this API, the client must first establish a connection with the server in order to perform operations. This is done via an HTTP request, used for establishing the WebSockets connection. In order to test the provided API and the capabilities of the framework, we also devised a client which performs the operations, available on [43].

When connected, clients and servers exchange JSON formatted messages that contain messages that trigger two types of operations: the first is a **request**, which is similar to an HTTP request, where the client creates a request and assigns it an ID which it sends to the DeMMon server via a WebSockets message. When the server receives the message, it processes it and sends back to the client a reply message containing the ID and the reply contents using the same established connection. Requests are used, for example, for querying metrics. The second type of operation is a **subscription**, which initially performs similarly to a request, however, in this operation, the server, posterior to the initial request, may send messages to the client (without requiring a request) containing events related to the issued subscription. This type of operation is used by clients for receiving events, for example, for both clients wanting to receive active view changes or for clients installing alarms and then receiving updates to changes in these alarms.

With this, we now provide a brief overview of the more relevant operations exposed by the DeMMon API.

4.5.1.1 API operations

1. **Install or remove buckets** These operations, as their name indicates, insert or remove buckets from the time series database. As previously mentioned in subsection 4.4.0.2, buckets are containers for all time-series data with a certain name, periodicity and capacity. Whenever a "create bucket" operation is issued for a bucket with a name that collides with a pre-existing bucket (with a different periodicity), an error message is returned to the client.

2. **Retrieve and insert metric values**. These two operations, performed via requests, add or extract values from the time series database. When these requests are received, the values are inserted directly in the corresponding time series. The retrieval of metric values is also performed via a request containing a query in the query language. This query is passed to the monitoring module (specifically the

metric engine) for processing. When the query has finished being processed by the metric engine, the result is sent back to the client via a message.

3. **Subscription to active view updates**. This operation, as the name denotes, is performed via a subscription, where the initial reply contains the current view of the server. Then, whenever there is a change in the active view of the DeMMon server's overlay protocol, the client is sent a message containing the changes in the active view.

4. **Install continuous query**. This operation allows the installation of a continuous query. Continuous queries is a feature that, when installed, evaluates a query at a specified periodicy and inserts the return values into the time series database under a specified name. This operation is useful for applications that wish to, for example, resample their data (i.e. to longer periodicities) or wish to calculate a certain aggregated value at specific intervals.

5. **Send and receive broadcast messages**. As the name indicates, these two operations refer to issuing and receiving broadcast messages. The emission of new broadcast messages is done via a request containing the message contents, the message TTL and the message ID (a text-based field). Whenever this request is received, the API sends a request containing the supplied message to the overlay protocol, which, in turn, propagates the message contents via its active connections (until the message TTL is 0). Broadcast messages are propagated and forwarded to peers in the active in a similar manner to the subscription messages (described in 4.3.2). Finally, nodes can perform **Subscriptions to broadcast message receptions**, which are subscription operations for clients that wish to receive all messages with a certain ID that are received by the DeMMon overlay protocol.

6. **Install Alarm**. This operation is performed via a subscription containing the parameters described in Section 4.4.0.4, whenever a client issues this operation, the API assigns a new ID to the alarm and adds it to the alarm manager, where it is periodically verified. After this, if in any verification of the alarm fires, the alarm manager notifies the API, which correspondingly sends a message to the client with the firing alarm's ID.

7. **Install and removal of neighbourhood aggregation set**. These operations manage the operation of the neighbourhood aggregation algorithm defined in Section 4.3.2. Whenever a neighbourhood aggregation set is installed, it is assigned an ID and a reply is sent to the client with the generated ID. When the client wishes to stop the collection of these values, it issues a removal request containing the originally assigned ID.

8. **Install and removal of global aggregation function**. These operations initiate and stop the global aggregation procedure (described in subsection 4.3.3), the behaviour

of this operation, in regard to the interaction between the client and the server, is similar to the neighbourhood aggregation set.

9. **Install and remove tree aggregation function**. This is the last detailed operation of the deMMOn API, and triggers the operation of protocol 4.3.1 which, in terms of interaction between the client and the server, also behaves similarly to both the neighbourhood and global aggregation requests.

4.5.1.2 Summary

In this section, we have presented an overview of the operations exposed by the DeMMon API. We began by providing a brief explanation behind the choice of WebSockets [49] as a way to enable message-based communication between the API and the client. Next, we enumerated what we believe to be the most relevant operations of the API, and for each enumerated operation, we provided a brief explanation of its behaviour regarding both the effect on the remaining components and the model of interaction between the client and the server.

4.6 Summary

In this chapter, we covered the implementation of the DeMMon framework, a decentralized management and monitoring framework targeted for the operation of decentralized resource management systems. We began by covering what we believe to be the requirements of this solution (enum. 4). Following, we provided a brief overview of the four modules which compose this framework, beginning with the overlay network (sec. 4.2), which is responsible for creating and maintaining a multi-tree shaped network, optimized using latencies and node capacity. Following, in Section 4.3 we covered the aggregation protocol, which provides multiple primitives for collecting and aggregating metrics in a decentralized and efficient manner, using in-transit aggregation, from a partial (or complete) set of nodes in the tree-shaped network. Next, we covered the monitoring module (sec. 4.4), which is the module responsible for enabling the storage and retrieval of metrics, parsing and processing queries and managing alarm lifecycles. Finally, we finished the DeMMon implementation by covering the API (sec. 4.5), which essentially is the module responsible for mediating, via a WebSockets interface, the aforementioned interactions between the external clients and the other modules.

5

PouchBeasts: A Benchmark Application

In this chapter, we present the third contribution from this dissertation, named "PouchBeasts". PouchBeasts consists of a benchmark application, for a backend of an edge-enabled interactive multiplayer game, with the functionality inspired by the popular game PokemonGO [8]. This contribution arose from the suggestion presented in [35], and it aims to present the materialization of a benchmark with a focus on real-time interactions between users. The importance of this contribution is its' possible use to test service deployment systems, given the user interactions can be dramatically influenced, in terms of quality of service, the proximity to the deployment of its' services (i.e. users performing real-time battles mediated by a server in a different continent will have a poor user experience).

PouchBeasts was attained through the combined efforts with a colleague, with the goal of being a proof-of-concept for the realization of a fully decentralized resource management system. The intention is to use this benchmark with DeMMon as the solution for managing the nodes in an overlay network and for monitoring the execution of the PouchBeasts microservices. Then, this monitoring information is used by a decentralized service deployment solution to optimize the services supporting the execution of "PouchBeasts", via heuristics based on geographical position, the latter being our colleagues' work.

In this application, registered users own a set of beasts, which they can expand by catching more beasts in certain geographical areas or by acquiring new ones in a shop. Beasts are collectable items with different properties (such as attack value, health points, experience, among other properties) that may be used to both battle against other users (and their beasts) and join other users in a cooperative battle against a computer-controlled beast. During these battles, users must command their beasts to either attack or defend and can also use items on their beasts, which can have multiple effects, such as: reviving dead beasts, healing a certain amount of health of a beast, among other uses. These items may be traded with other users or acquired from a shop using coins, which in turn are acquired through microtransactions.

5.0.1 Overview

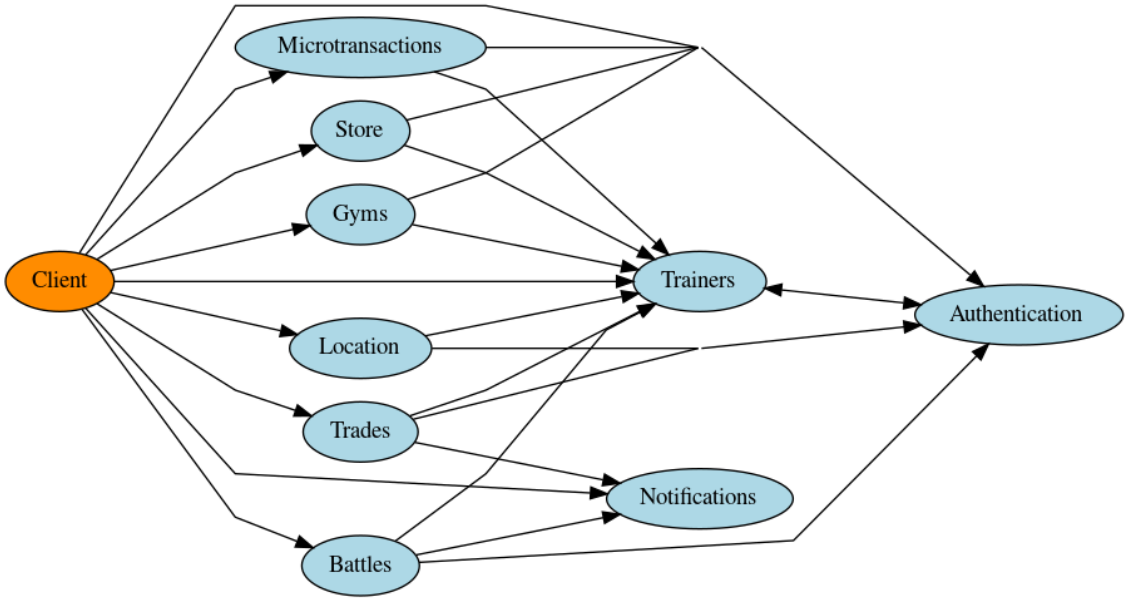

Figure 5.1: An illustration of the dependencies between services of PouchBeasts

The interactions between the services in this benchmark are illustrated in Figure's I.4 diagram. There are nine microservices in total and a client to access them. We now provide a brief overview of each microservice and its role within the system:

1. The first and most used microservice of the system is called **Trainers**, and it essentially stores all the data related to the users and their owned beasts. In addition, the service verifies the tokens issued by the Authentication microservice in regard to the recency of the information carried by the tokens. This service makes use of a MongoDB [61] database to store these records in permanent storage and to maintain data consistency across microservices.

2. The next microservice is called **Authentication**, which only has the purpose of generating new authentication tokens for the users to use when interacting with other services. These tokens contain a hash of the owned beasts, so other servers can verify their authenticity and recency without having to fetch the users' beasts on each interaction.

3. Following, we have the **Gyms** and **Battles** services and these allow players to perform combats with their obtained beasts. In the case of the Gym's service, it manages entities in the system denominated gymnasiums, which have a pre-assigned geographical location. In this service, if a user is within a geographical distance of a gymnasium, it may perform battles alongside other trainers against a single beast controlled by the computer. The **Battles** service is a service that allows users to

use their beasts to perform battles against other users' beasts. Battles can either start via a queueing system, where players wait for another random user to start the battle or by challenging other known users (via a notification). Whenever a user is in a battle, it issues commands (based on the observed battle status) and receives updates regarding the status of the battle. As commands depend on the observed status of the battle, it is paramount (for the quality of service of the users) that both the players' commands and the information passed from the battles/gyms service to the player suffer the least latency possible. The information regarding the issued commands and battle status is propagated to the user using WebSockets. Whenever a battle finishes, the Battles / Gyms server stores the battle result and update the users' beasts and items in the Trainers service.

4. The **Store** and **Microtransactions** services provide ways for users to obtain currency via small value transactions, which can, in turn, be used in the store to buy new items. These items then have effects on the beasts (e.g. or reviving a dead beast or healing a beast with low health).

5. Users may also change their items via the **Trades** service. This service grants users the possibility to exchange their items with other users in the system. In order to use this system, a user must invite another currently active user via a notification (which can optionally be accepted by the target user). Whenever this notification is accepted, the two users connect to the server via a WebSockets connection and begin to submit the items they wish to trade. Whenever a player adds an item to the trade, this information is propagated by the server to the other player through a WebSockets connection. Whenever the users finish adding or removing items to the trade agreement, they accept the trade, and the Trades server commits the transaction result to the Trainers server.

6. The service responsible for handling all of the previously mentioned notifications is called **Notifications**. This service is essentially tasked with receiving notifications from connected users and propagating them towards the target user. As there may be multiple Notification services executing concurrently and users may connect to any of the available servers, a notification may be emitted for a user not connected to the same server. To prevent these notifications from being lost, this service makes use of a Kafka [2] backend, which it uses to propagate messages targeting users that are connected to different notification servers.

7. The last implemented microservice is called **Location** service, responsible for tracking the following entities contained within a geographical area: the geographical locations of users, the generation and management of the generated beasts (for users to catch), and the locations of gymnasiums. Essentially, location servers communicate with the users through a Websockets API, where users periodically receive locations of the nearby beasts and gyms.

In order to prevent multiple location services from managing overlapping geographical areas and to facilitate the insertion and decommission of new location servers, we assign portions of the geographical area to certain servers using S2 cells [53]. S2 cells provide a framework for decomposing a sphere (in our case, the earth) into a hierarchy of cells, where each S2 cell is quadrilateral bounded by four geodesics. The top-level of the hierarchy is obtained by projecting the six faces (the topmost six cells) of a cube onto the earth, and lower levels are obtained by subdividing each cell into four children recursively. An example of two of the six face cells (one of which has been subdivided multiple times) can be observed in Figure 5.2. This service makes use of S2 cells to (1) assign portions of the earth to servers in a way that does not create geographical discontinuities, (2) to index efficiently the locations of trainers, gyms and generated beasts, which allows the service to, based on the intersection of a cell centred on the user's location with the cells indexing the gyms and beasts, determine the set of beasts and gyms to return to emit to a certain user, (3) in the case a user's location is in the boundary of two (or more) location servers, S2 cells are also used to decide to which server(s) the user should connect. Similarly to before, this is also performed based on the intersection of a cell centred on the user's location with the cells indexing the gyms and beasts.

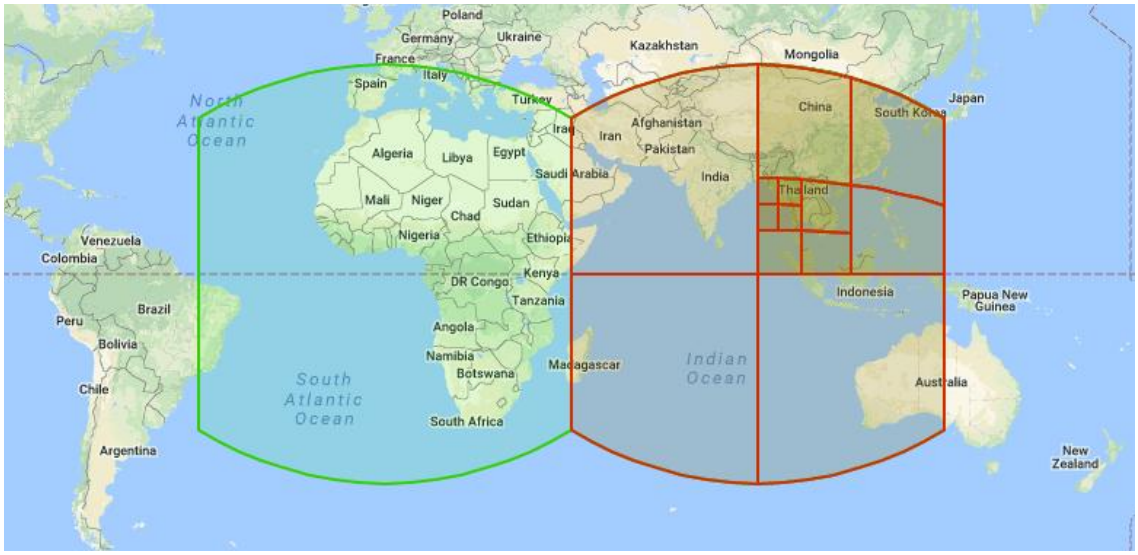

Figure 5.2: Example of S2 cell hierarchy, taken from "https://s2geometry.io/"

Provided with the high-level overview of each of the implemented microservices of the benchmark, it is important to notice that the latency requirements regarding the interactions with the users and services are varied. For example, services such as the Trainers, Store or Microtransactions services are more tolerant when it comes to latency requirements when compared to services like the Gyms, Battles or Trades, as these have an interactive nature where a high latency value leads to a worse user experience.

To test these interactions, the benchmark also contains a client which allows the

execution of actions such as: battling other users, catching beasts, acquiring and spending tokens, among others. Then, through the instrumentation of both the client and the services, we provide metrics to quantify some aspects regarding these interactions, such as the delays in the interactions between users and servers in both Trades and Battles services, among other interactions. Provided with these indicators, then the performance of service deployment and maintenance systems can be assessed.

To enable automated client testing, we provide ways for clients to simulate user behaviour. This is configurable via a configurable stochastic matrix, that contains a line and a row for each possible action to perform with the client, and each matrix position (given by a certain line and row) contains the probability of performing any of the other possible actions, provided the user just performed the action in the current line.

Provided the objective of this benchmark is to test service deployment systems, we provide a deployment configuration of this benchmark for Kubernetes. With this baseline deployment system, commonly employed in the state-of-the-art, users who develop their service deployment systems have a baseline to compare to.

5.0.2 Summary

In this chapter, we covered the third contribution from this dissertation, named "PouchBeasts". This contribution, in the shape of a benchmark, aims to simplify the evaluation process of decentralized management and monitoring solutions, particularly those aimed at improving service deployments. It does so by providing both a client and a set of services (implemented by microservices) that offer a wide range of interaction types, from request-reply based interactions to real-time interactions, with varied demands in regard to server and client latency.

Although this benchmark was not employed to test the performance of DeMMon directly, it is important to mention that the previously mentioned colleague, that contributed the implementation of this benchmark has successfully built a system (for his dissertation) that, through the metrics obtained by the decentralized aggregation primitives provided by the DeMMon framework, improves the QoS of clients using the "PouchBeasts" services by placing the services geographically closer to clients.

6

Evaluation

In this chapter, it is our goal to demonstrate the applicability of the devised solution through the comparison of multiple aspects of the framework against popular baseline solutions from the literature. To this end, Section 6.1 covers the experimental setting and configuration in which these comparisons were conducted, namely the execution environment, the specifications of the nodes executing the tests, among other aspects of the experimental setting. Following, in Section 6.2, we provide the obtained results from the conducted experiments testing the applicability of the devised protocol against state-of-the-art baselines. We compare these baselines with our protocol both in their capacity to create and establish the network and in their capacity to perform information dissemination. Lastly, Section 6.3 covers the experimental evaluation of the implemented aggregation protocol, notably, which solutions composed the baseline for comparison, which experiments were carried, and the obtained results.

6.1 Experimental Setting

To conduct the experimental evaluation of the devised solution against state-of-the-art baselines, instead of resorting to simulation, we implemented those solutions and tested them in an emulated network that aims to be as similar as possible to real-world scenarios. Provided that scalability is one of the components we aimed to test, and there is a limited pool of individual machines in our testbed to conduct the experiments, we resorted to using containerization. Containers allowed us to execute multiple independent processes in a single physical node while still being an isolated environment that allowed the manipulation of the networking conditions of each process.

As containers are running in different machines, without any additional software, a container from a machine would not be able to communicate with containers executing in a different machine. To overcome this, we made use of Docker [15] containers and employed a tool called docker swarm [59]. This tool allows users to coordinate sets of nodes running Docker. Nodes in a swarm, among many other features, may perform Multi-host networking, which consists in integrating the containers executing among

nodes running into a unified network. In this network, containers are automatically assigned IP addresses and can communicate with each other, regardless of the physycal machine each container is executing on. To bootstrap the experimental scenario, we developed a set of scripts in both BASH, Python and GO to create, orchestrate, and decommission containers to run the experiments.

6.1.1 Node capacity and connection delays

As previously mentioned, in order to emulate a real-world scenario where nodes have limited capacity and their connections have delays, there was the need to apply these constraints. Furthermore, it is important to set up these delays realistically. To do so, we used data from real-world readings of real-world scenarios obtained from WonderNetwork [45], which consists in a network composed of 252 nodes spread across 88 countries in 6 continents. This network provides, in addition to node metadata (city, country, among others), a set of latency measurements from each node to every other in the network (including themselves), which we used to setup the latency delays between the containers. As there was the need to test the framework with larger network sizes (of up to 750 nodes), the data points from this network were multiplied by five times.

Then, as the obtained data from this network did not contain bandwidth information for each node, we used the metadata provided by WonderNetwork, namely the country, to assign bandwidth values according to the list of bandwidth per country provided by speedtest.net [27]. Provided the purpose of this framework is to perform on cloud-edge scenarios, composed by nodes inside and outside of the data-centre (DC), where nodes outside the DC have lower networking capacity comparatively to nodes running inside the DC, we divided each data point by 12x (to represent the edge nodes running outside of the DC), and divided the first N nodes by 2.5x, (corresponding to the number of data centres).

To limit the networking capacity and inject latency in the containers executing the protocols for the experiments, we used a tool called Traffic Control (TC) [3]. This tool is a traffic shaping tool that performs shaping, scheduling, policing and dropping of network packets through the configuration of the kernel packet scheduler.

In our case, we used this tool to limit both the available inbound/outbound bandwidth on the container interfaces and to inject delays in all connection pairs. In order to limit the available bandwidth, we used hierarchical token buckets (HTB), which are classful queueing disciplines that employ a complex token-borrowing system to ensure the shaping of traffic according to a (configurable) rate. HTB requires programmers to set up a hierarchical class structure, where child classes, attached to a queue (or qdisc), manipulate packet order and apply rate limiting policies according to configuration.

For our benchmark, we made use of rate-limiting policies, which employ a token borrowing mechanism that functions in the following manner: whenever a certain child class reaches the maximum of its rate, it borrows tokens (up to its **ceiling** value) from the

parent class (if there is a parent class, and that class has available tokens). If the parent class is also limited, then the sum of its child classes will be limited to the parent rate.

For our experimental configuration, each container creates two default qdiscs, attached to the inbound and outbound networking interfaces. Then, two HTBs are attached to these qdiscs and set up with the respective inbound and outbound bandwidth rate. After this, for both the outbound and inbound classes, two child classes are installed: one intended for latency measurements and keepalive traffic (specific UDP traffic on a pre-configured port); and the other for the remaining traffic. The inbound and outbound classes responsible for measurement traffic are assigned a fixed rate of 500kb, and the inbound and outbound default traffic classes are assigned the remaining rate for the container (according to speedtest.net) minus the 500kb rate for the measurements class. Then, for all the outbound classes (measurement and default traffic), we set up another set of HTB classes for each other container with a very low rate of 6kb and ceiling rate corresponding to the parents' class. This setup forces child classes to borrow tokens from the parent class and be limited by its bandwidth rate.

For each of these leaf classes, we attached a netem qdisc. This qdisc applies a delay to each packet according to the latency measurements taken from WonderNetwork. To route packets from one qdisc to the other, we use filters: in the case of the measurement traffic, the filtering was performed via installing a high-priority filter that verifies the source and destination ports of the packets and sends it to the measurement classes. The remaining traffic is forwarded to the default traffic class via a lower priority filter with no restrictions. After this, the routing from these two outbound classes to the leaf classes is performed via filters observing the destination IP address of the packets and redirecting them to their corresponding netem qdiscs. The objective of separating the traffic into two distinct classes with their own bandwidth values is to prevent cases where the applicational traffic is high (i.e. testing information dissemination) and the delay caused by the high usage of the data channels would interfere with the measurement packets, leading to incorrect latency measurements and consequent instability during experiments for both DeMMon and the baseline overlay protocols that optimize the overlay.

Experiments presented in this work were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). The hardware from this testbed used to carry the experiments consisted in sets of 10 physical nodes for the experiments with 50 and 250 logical nodes and sets of 30 physical nodes for experiments with both 500 and 750 logical nodes. Each of these physical machines is equipped with 2 x Intel Xeon E5-2630 v3 and 128 GiB of RAM and is executing Linux Debian version 4.19.104-2 and Docker version 20.10.7. The results were obtained through logging the relevant aspects of the experiment to disk and then processing the obtained logs to extract the intended information posterior to the end of the experiments.

Provided with the experimental setup, we now explain the steps taken and the results obtained for the DeMMon framework evaluation.

6.2 Overlay Protocol

In this section, we present the results obtained from the multiple conducted experiments of the overlay protocol against state-of-the-art baselines. These experiments aimed at testing: (1) the cost of establishing/maintaining the overlay networks for each protocol; (2) testing the established networks' efficiency (according to latency); and (3) finally, testing DeMMons' message dissemination capabilities against that same set of baseline protocols. In this last test, the baseline membership protocols are paired with two distinct dissemination protocols to perform the message dissemination. We now begin by providing a brief discussion of the protocols and parameters used for conducting the experiments.

6.2.1 Baselines and configuration parameters

The chosen protocols to perform the overlay protocol network comparison were **Hyparview** [32], **X-Bot** [36], **Cyclon** [**voulgaris2005Cyclon**] and **T-Man** [28]. We now provide a brief description of each (further discussion is provided in Section 2.2).

Hyparview is a protocol that builds a non-structured overlay network using a fixed-sized view materialized by active bidirectional TCP connections (these connections also serve as fault detectors).

The second baseline protocol is **X-Bot** [36], which is a protocol that essentially behaves similarly to Hyparview in terms of establishing the initial network structure but takes iterative steps to optimize the overlay network (according to a configurable heuristic). These steps are performed via gossip mechanisms to improve the nodes' active connections' costs. In X-Bot, nodes perform optimizations in such a way that maintains the in and out-degree established initially.

The third implemented baseline protocol was **Cyclon** [**voulgaris2005Cyclon**], which is an overlay protocol that materializes a network composed of asymmetric links via periodic exchanges of node pointers with a configurable age.

The last implemented baseline was **T-Man** [28], which is a protocol that iteratively builds on an existing set of nodes to build a new, more optimized set of nodes. These optimizations are performed iteratively by each node in the system such that a configurable cost function (defined a priori) gets minimized. In order to feed the initial node sample for T-man to optimize, we employed the Cyclon protocol, and consequently, the evaluation results for this protocol are labelled as "Cyclon T-Man". All of the described baseline protocols were, for comparativeness, implemented using the GO-Babel framework (described in Section 3.2). As Go-Babel only provides unidirectional connections, protocols that require bidirectional connections, namely Hyparview, X-Bot and DeMMon, were enriched with added mechanisms to ensure that the bidirectional connections were established for each node contained in nodes' active views.

The utilized parameters for the different protocols were tuned to attempt to perform a fair comparison. These are displayed in table 6.1, we now describe each of the columns.

Table 6.1: Membership evaluation: protocol configuration parameters

	VSizeMax	VSizeMin	PVSizeMax	Shuffle δT (s)	PRWL	ARWL	ka	kp	improvement δT (ms)	UN	PSL
Hyparview	5	-	25	5	6	3	2	3	-	-	-
X-Bot	5	-	25	5	6	3	2	3	50	1	2
Cyclon	7	-	-	5	-	-	-	-	-	-	-
Cyclon T-Man	5	-	7	5	-	-	-	-	-	-	-
DeMMon	5	2	25	5	6	-	-	-	50	-	

The first is called "VSizeMax", and represents the maximum size of the active view, which in most protocols was set to 5, except for Cyclon, where it was set to 7 as it is the only protocol without a secondary backup view. In the case of DeMMon, "VSizeMax" represents the maximum number of children per node.

The second column, named "VSizeMin", corresponds to the minimum number of children for each node in DeMMon. The third parameter, titled "PVSizeMax", corresponds to the maximum size of the passive view, which is set to 25 for DeMMon, Hyparview and X-BOT, and set as 7 for the case of Cyclon T-Man. In the case of T-man, this parameter corresponds to the size of the Cyclon view (running to provide its initial view). The next parameter, labelled "Shuffle", corresponds to the periodicity of each protocols' shuffle mechanism. This parameter is set to 5 seconds for all protocols. Following, we have the "PWRL" parameter, which corresponds to the TTL of the random walks (for each protocol that has a random walk mechanism). The last-mentioned parameter is the "δT" parameter, which corresponds to the minimum latency improvement for both X-Bot to perform active view exchanges and for DeMMon to make opportunistic improvements. Some parameters such as timeouts and the duration of some periodic procedures were omitted, however, all timeouts, e.g. timeouts for dialling nodes, receiving message responses, among others, are set to 5 seconds. Furthermore, all periodic mechanisms are executed with a frequency lower than 15 seconds.

6.2.2 Overlay construction and maintenance

The first conducted experiment, aimed at evaluating how protocols establish and maintain the overlay network, consists of an experiment where different numbers of nodes join the system and remain for 25 minutes. In this experiment, we evaluate the properties of the built overlay networks (costs, degree distribution, among other properties) and how fast the protocol converges towards an optimized network. Finally, to compare the scalability, performance and fault-tolerance at multiple scales, we performed the previously mentioned experiment using network sizes of 50, 250, 500 and 750 nodes along with two failure rates of 0 and 50%.

In the graphs displayed in Figures 6.1 and 6.2, we may observe the results pertaining to the average latency of a connection in the overlay and the total cost of the established overlay networks for the experiment with no failures, respectively. For both of these graphs, we show the results obtained from both the baseline protocols and the DeMMon protocol. In the case of DeMMon, we make a distinction between two latency values,

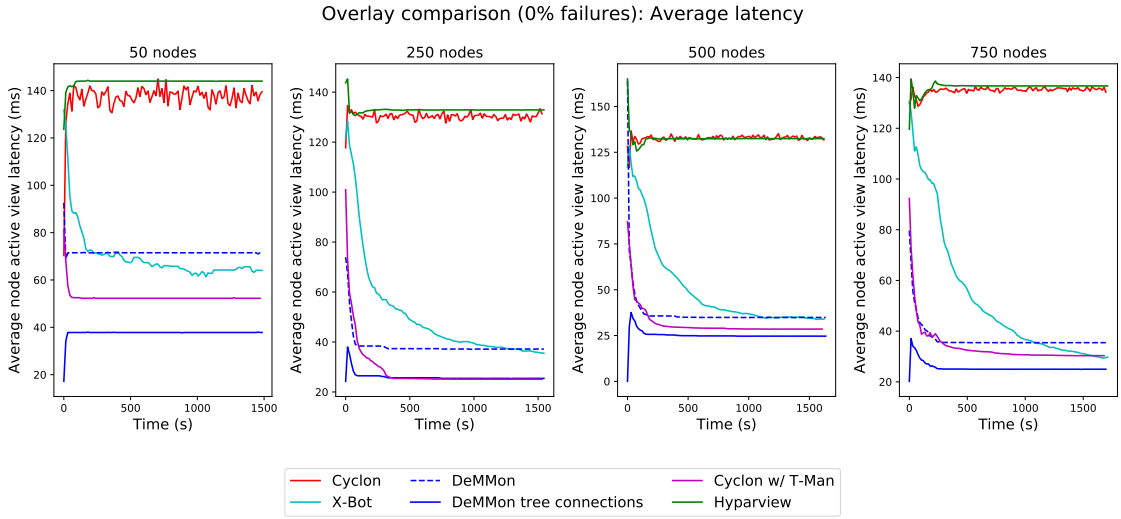

Figure 6.1: Average latency per node in established networks

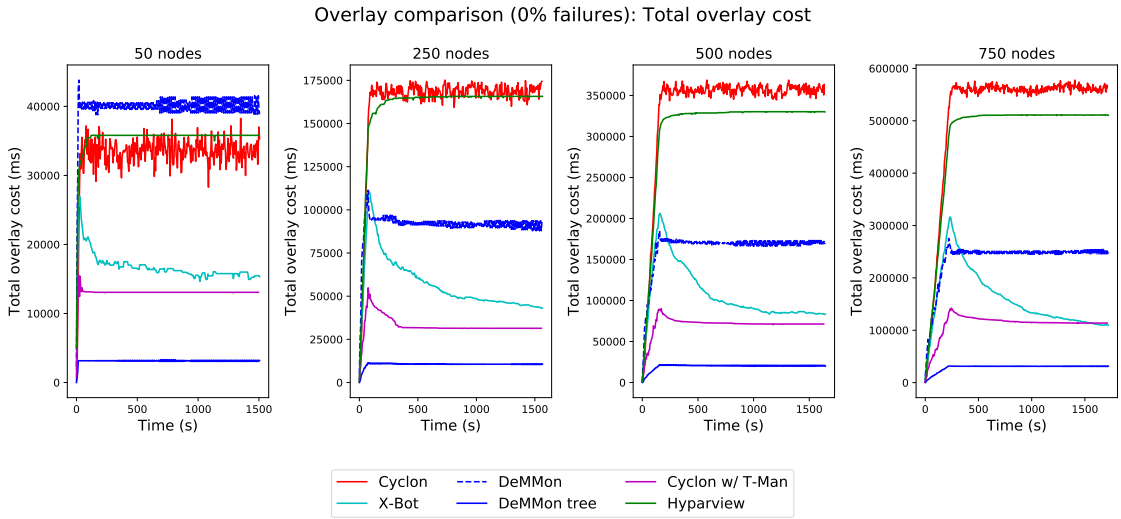

Figure 6.2: Total network cost (in latency)

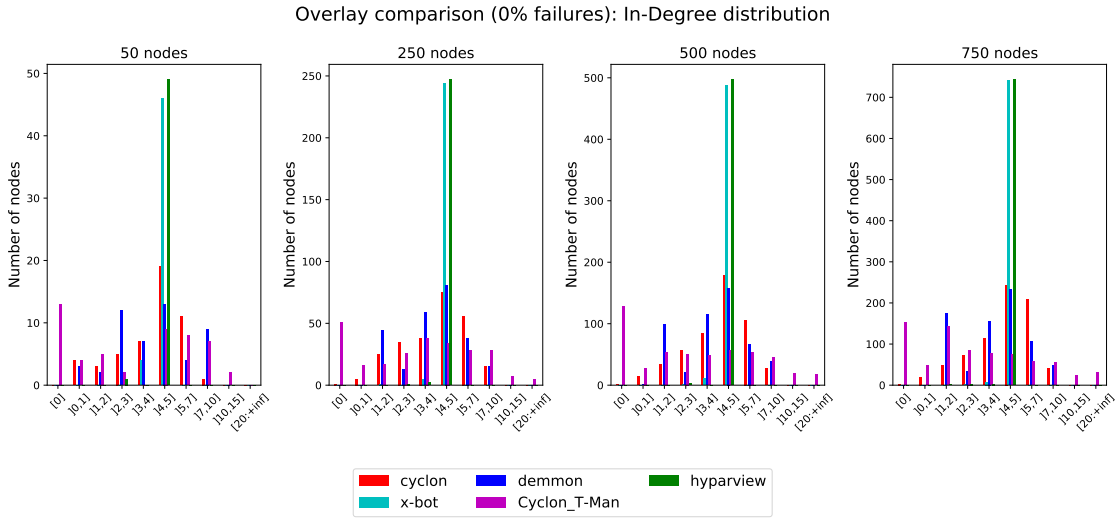

Figure 6.3: Node in-degree

the first (represented by a blue, continuous, line) represents the results relative to all connections of all nodes, the second value (represented by a blue dashed line) represents the cost of the "vertical" connections of the DeMMon tree (the parent and children of each node), essentially excluding the siblings of each node from the results. We made this distinction for two reasons: first, as the DeMMon protocol only performs optimizations to improve the parent connection, we believe it is important to see the correlation between improving the parent connections to the sibling latencies. The second reason to make this distinction is that these connections are more used when compared with the sibling connections, namely for network maintenance, information dissemination and in-transit aggregation. The results displayed in these graphs (6.1 and 6.2) show that both Hyparview and Cyclon converge to a similar average latency value, which corresponds to the average of all connections of the latency matrix. This is expected, as these protocols do not attempt to perform optimizations in regard to the network latency. The results also show that the devised protocol is the fastest to converge to its lowest latency value and that X-Bot is the slowest, not converging to a final value in a test of 25 minutes. We believe this occurs because X-Bots' overlay improvements are performed using 7 messages, contrasting heavily with DeMMons' 2 required messages, and T-Mans' 0 required messages. While the total and average latency of the DeMMon overlay is not the lowest in any of the displayed results, when comparing only the parent and children connections, DeMMon reaches a total latency cost lower than any other tested protocol. This is important given that, as previously mentioned, these connections are the ones most used when performing overlay improvements and maintenance, information dissemination and in-transit aggregation.

It is important to mention that, while T-Man is the protocol that reaches the lowest overall and average latency in the conducted tests, it does so disregarding the fact that

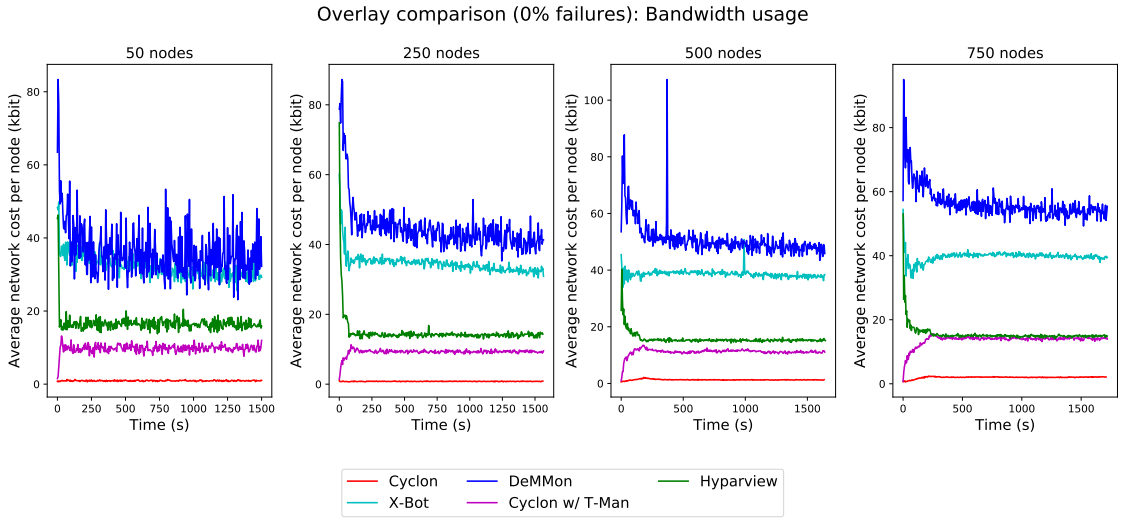

Figure 6.4: Protocol bandwidth cost

nodes may become disconnected from overlay, which as we will observe further, prevents this protocol from being suitable for reliable message dissemination. This may be observed in fig. 6.3, which shows the in-degree (the number of incoming connections for each node) for all nodes participating in the network, these results pertain to the last observed configuration of the network before the experiment finished. They show that T-Man, at multiple node counts, possesses nodes with 0 incoming connections, which are effectively isolated from the network. While still analyzing the in-degree results, we observe that both X-Bot and Hyparview have a fixed number of incoming connections, which stems from the use of bidirectional connections, while Cyclon has varied numbers of incoming connections ranging from 10 to 1, which occurs due to the shuffle mechanisms of the active connections. In the case of DeMMon, the values range from 2 to 10 incoming connections, which is expected given the configuration parameters of a minimum number of children of 2, and a maximum number of children of 5.

Finally, still regarding the experiments without node failures, we show, in Figure 6.4, the average network cost (in kbit/5s) incurred by each node running the experiments. This graph shows that DeMMons' overlay protocol, on average, spends more bandwidth to build and maintain the network structure, we believe this is because DeMMon exchanges more information periodically with peers in the active view to maintain and improve the network structure when compared to the other protocols. Conversely, the protocol which uses the least amount of bandwidth is Cyclon, as its shuffle mechanism is relatively inexpensive, and the protocol has no other mechanisms that incur networking costs. Although protocols have varied networking costs, we believe that even DeMMon, which uses the most bandwidth, is relatively inexpensive when compared with the bandwidth standards at the time of writing this work.

Provided with the result analysis for the experiments with no failures, we now provide

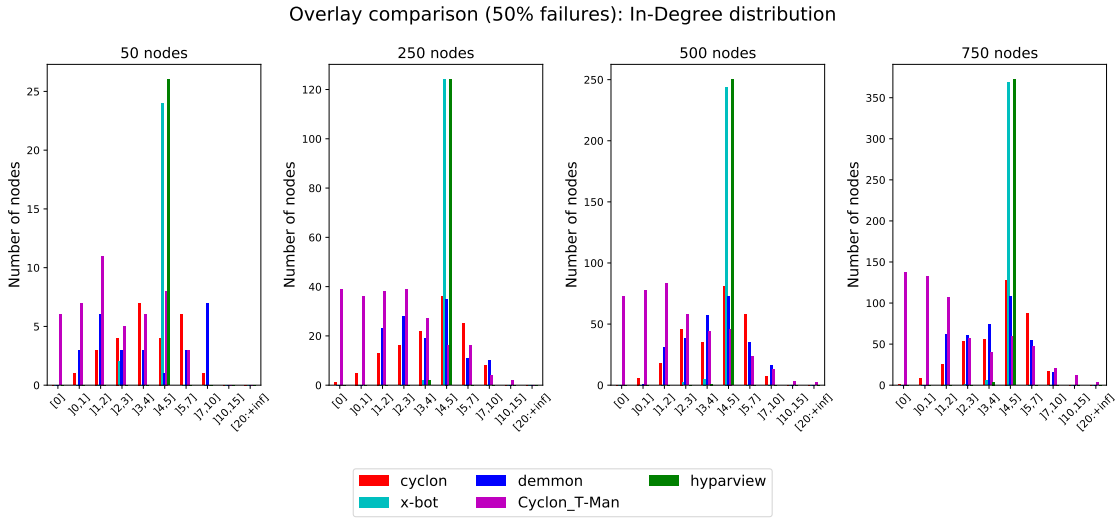

Figure 6.5: Node in-degree (50% failures)

the results for the in-degree distribution of the protocol in a scenario with failures. This experiment attempts to test the fault tolerance of the protocols by first establishing the network, and during the middle of the experiment, induce a failure of 50% of the nodes. The objective of this experiment was to test if any node became isolated from the network after the failures. Results from this experiment may be observed in Figure 6.5, where it is observable that, for all tested protocols except T-Man, no nodes became isolated, allowing us to conclude that both the devised protocol and the tested baselines can recover from faults effectively.

As previously mentioned, the applicability of our solution was tested in two different aspects: the first was the process of building and maintaining the overlay network, which was covered in the previous paragraphs. The second evaluated aspect is information dissemination (via message broadcasting), which we will now cover in the following subsection.

6.2.3 Information dissemination

The second set of conducted experiments, as mentioned previously, intends to test the applicability of the devised membership protocol in an information dissemination scenario. To do so, we tested it against the same set of baseline protocols used in the previous experiments enriched with two message dissemination protocols: the first is a simple flood protocol, where if a node wishes to broadcast a message, it sends that message to every peer in its active view, then, nodes that receive this message, propagate it to every neighbour if they have not done so previously (excluding the sender). The second used dissemination protocol was PlumTree [33], which is a dissemination protocol that builds a dissemination tree based on the paths taken by the broadcast messages.

The reasoning behind this choice of dissemination protocols was to provide a more comprehensive comparison of DeMMon with the remaining protocols. As the simple flood generates redundant messages when compared to dissemination primitives tree structures, we included a dissemination protocol that (similarly to DeMMon) also employs a tree for the dissemination of messages. It is important to mention that, when testing the PlumTree protocol, in order to establish the initial tree structure, a single node first starts the dissemination of its messages a minute earlier than other nodes. For both of these comparisons, DeMMon is set up with a dissemination protocol similar to the simple flood protocol, however only using its vertical connections (parent and children).

Similarly to the first set of experiments, we conducted multiple tests with 50, 250, 500 and 750 nodes during 15 minute periods. For all these system sizes, we also tested failure rates of 0 and 50%. For each of these combinations, we varied the number of messages each node emitted until all protocols reach their saturation point. While doing the tests, we extracted the following metrics: (1) the reliability of the messages, i.e. what is the average percentage of nodes that receive the emitted broadcast messages ; (2) the maximum message throughput reached by every protocol in a 30-second window, (3) the average latency taken by messages until they reach other nodes, and (4) the bandwidth usage of each of the protocols.

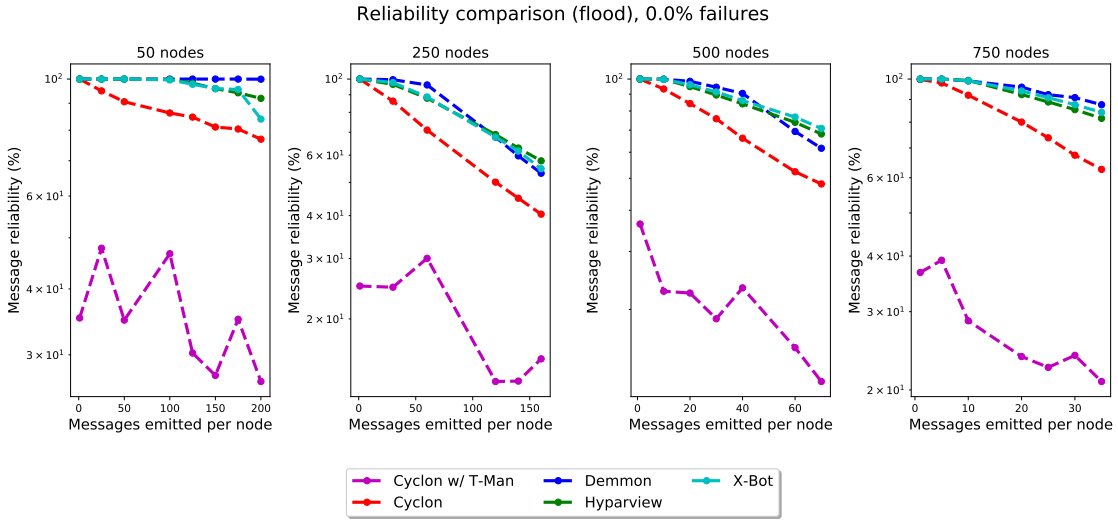

Figure 6.6: Average message reliability in simple flood scenario (0% failures)

Figures 6.6 and 6.7 show the obtained results regarding the message reliability during the experiments for both the simple flood and PlumTree experiments with 0 failures. As we may observe, in general, the saturation point for all protocols using PlumTree tends to be earlier (in terms of emitted messages per node) than the simple flood protocol. We believe this occurs because the PlumTree protocols' tree becomes unstable whenever certain nodes become a bottleneck to the messages being propagated using the tree (because their bandwidth capacity is exceeded). Whenever this occurs, as certain messages get

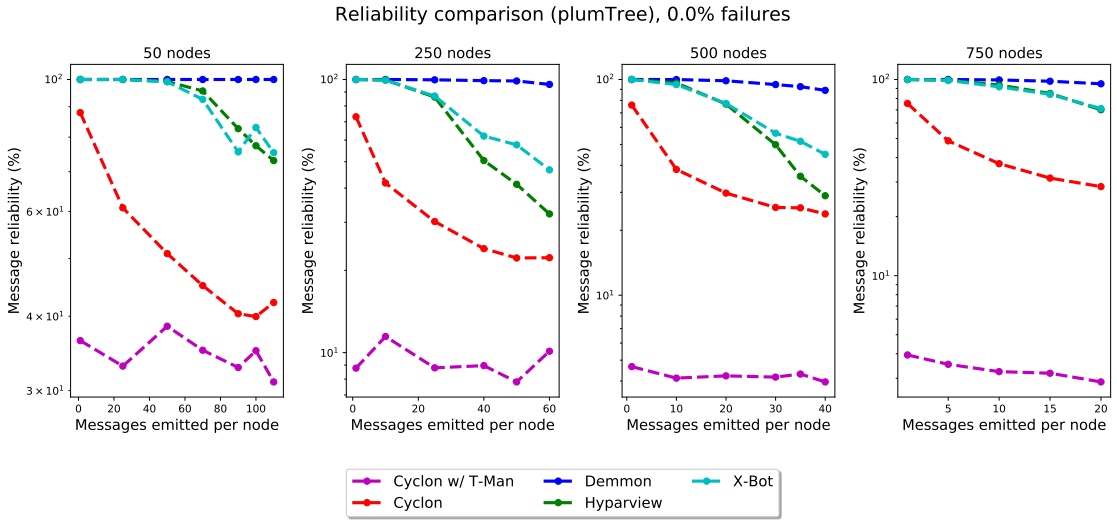

Figure 6.7: Average message reliability in PlumTree scenario (0% failures)

delayed, the tree structure becomes unstable (as the order of delivery of messages is what defines the dissemination tree structure). Whenever this occurs, the tree repair procedure is triggered, however as multiple nodes emit new messages and other nodes can become saturated while performing this mechanism, the tree structure may never reconverge until all messages are delivered (and nodes stop being saturated). Until this occurs, the protocol essentially becomes a push-pull gossip protocol, which has lower performance in our experiments in terms of reliability because message delivery requires 2 messages, incurring additional networking costs (and the tests end before the protocol has delivered all emitted messages).

In addition to the previously mentioned reasons, in an occasion where a node has received an IHAVE message for a certain message and at that moment happens to have available upload bandwidth, but its download capacity is all taken up by incoming traffic, this node will periodically emit GRAFT messages to the sender of the IHAVE message, which, in turn, will reply with the broadcast messages that will only be received after a large time frame. During this time frame, there may be multiple redundant GRAFT and IHAVE messages being emitted, which results in the system possibly becoming even more saturated, which causes the tree to become even more unstable. The devised overlay protocol, although it also uses a tree structure, its tree is not defined by the propagations of broadcast messages and consequently is not as susceptible to instability in conditions where the network is saturated, consequently achieving higher reliability in higher message counts.

We may observe that both the Cyclon and T-Man tend to perform worse in general regard to reliability when compared to DeMMon, Hyparview and DeMMon, which we believe, in the case of T-Man, to occur because there are nodes with 0 incoming connections and consequently do not receive any broadcast messages from other nodes. In the case of

Cyclon, we believe the lower reliability value is attributed to the use of UDP as its communication medium, which means that whenever the data channels become saturated, many of the broadcast messages are lost, contrary to DeMMon, Hyparview and X-BOT, that use TCP and consequently do not drop messages in congestion periods. Finally, we believe that both Cyclon and T-MAN, when paired with PlumTree, also have lower reliability because this protocol requires bidirectional connections to perform optimally, which are not guaranteed in either of these protocols.

In regard to the simple flood experiment (fig. 6.6), DeMMon tends to perform exceptionally well with fewer node counts, particularly with 50 nodes. We believe this may be due to the height of the DeMMon tree being lower, as when the tree height is smaller, the number of descendants for each node is fewer, which in turn means that when a certain node becomes saturated, fewer nodes are impacted by it. In higher node counts, DeMMon performs in line with both Hyparview and X-Bot. We believe this happens because the tradeoffs of having a tree (a single node possibly becoming a bottleneck for many other nodes in the system) tend to impact the system the same amount that sending multiple redundant messages does.

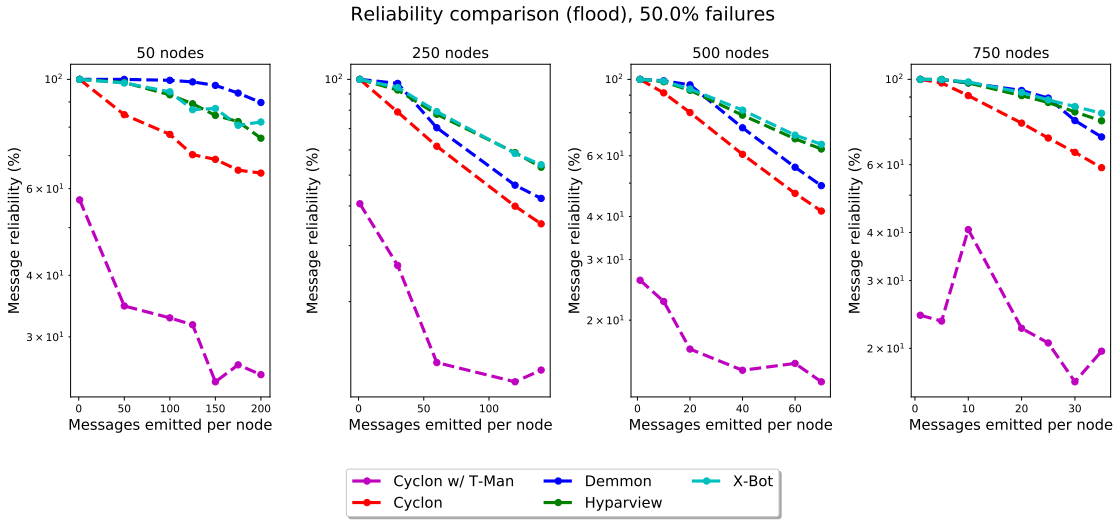

Figure 6.8: Average message reliability in simple flood scenario (50% failures)

In the case of scenarios with induced failures (Figures 6.8 and 6.9), we observe a similar trend in regard to the PlumTree experiments, with DeMMon achieving higher reliability values. However, in the simple flood experiments, we observe that DeMMon achieves a lower reliability value when under congestion, we believe this occurs because as the failures are occurring, if the nodes are saturated, the failure recovery mechanisms may take a long time frame to execute, and during this period nodes are disconnected from the remaining overlay and consequently do not receive or send message to or from any node which is not their descendant, leading to a lower reliability value.

Provided the results from the combination of the baseline protocols with PlumTree

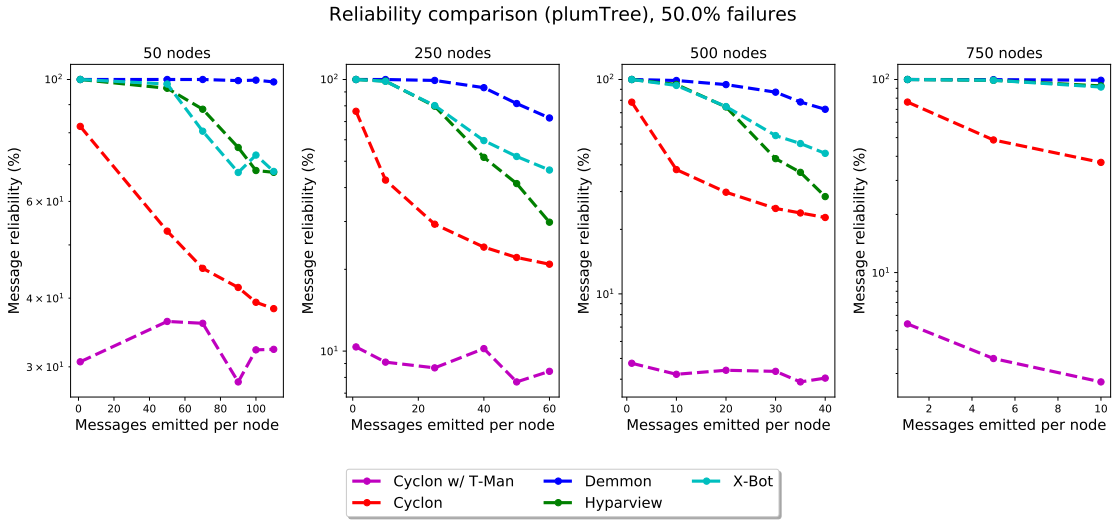

Figure 6.9: Average message reliability in PlumTree scenario (50% failures)

consistently performs worse in terms of reliability (when the network is saturated) when compared to employing only a simple flood protocol, we now focus on the comparison between DeMMon and the baseline protocols executing the simple flood protocol, however for completeness, all obtained results are available in annex I.

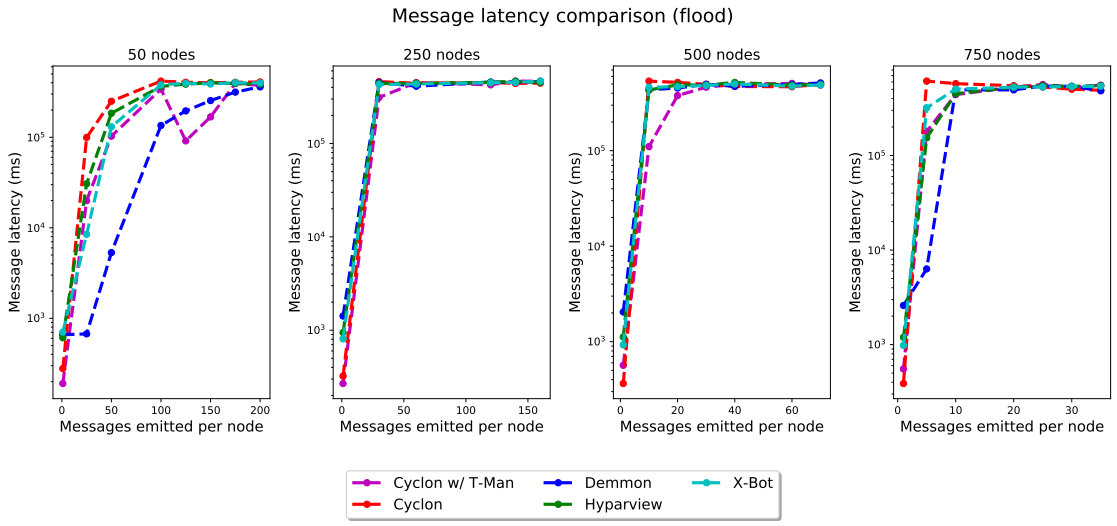

Figure 6.10: Average message latency (in ms) in simple flood scenario (0% failures)

In Figure 6.10, we may observe the obtained results from collecting the latency between the emission and reception of broadcast messages for each node. The first takeaway from these results is that all protocols plateau at the same latency value, we believe this is due to the fact the the test times are limited to 15 minutes, and whenever the system is saturated, all messages tend to take a similarly long time to be delivered, those which

are not delivered are only reflected in the previously discussed reliability graphs (Figures 6.6, 6.7, 6.8, and 6.9). However, for lower message counts, the latency results show that DeMMon tends to achieve lower latency values when compared with the baseline protocols on certain workloads (e.g. low numbers of messages emitted on both the 50 and 750 node graphs) where we believe the simple flood protocol becomes saturated due to the number of redundant messages sent.

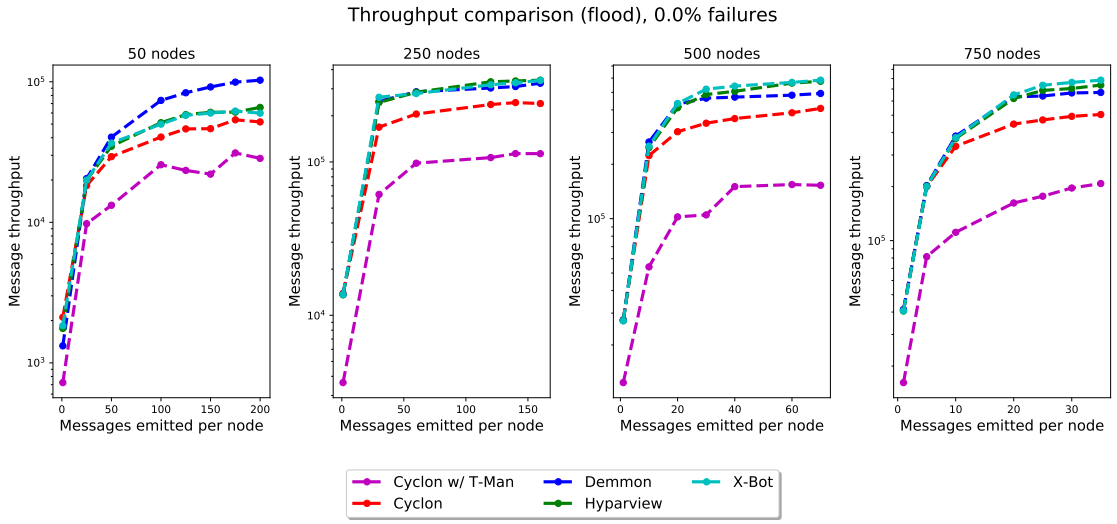

Figure 6.11: Maximum message throughput during experiment (30 second window) in simple flood scenario (0% failures)

In Figure 6.11, we may observe the obtained throughput across the message dissemination experiments for the simple flood protocol with 0 failures. As we can observe, in lower node counts (i.e. 50 nodes), the throughput achieved by DeMMon surpasses the throughput achieved by the remaining protocols, which also explains the higher values of reliability achieved by DeMMon in these node counts (see fig. 6.8). However, at higher node counts, all protocols tend to plateau at the same throughput, which we believe to be attributed to the fact that, as previously mentioned, the tradeoffs of using a tree (a single node possibly becoming a bottleneck for many other nodes in the system) tends to impact the system the same amount that sending multiple redundant messages does.

In Figure 6.12 we compare the baseline protocols with DeMMon in regard to the message latency. These results show the averaged latency distribution for the tests conducted with 250 nodes and one message emitted per node. On the left graph, we may observe the results obtained by the execution of PlumTree with the baseline protocols, while on the right graph, we have the results for the simple flood tests. As we can observe, in general, the message latency obtained by combining simple flood with the baseline protocols tends to be lower in latency when compared with protocols that employ shared trees to disseminate the messages, such as PlumTree and DeMMon. We believe this can

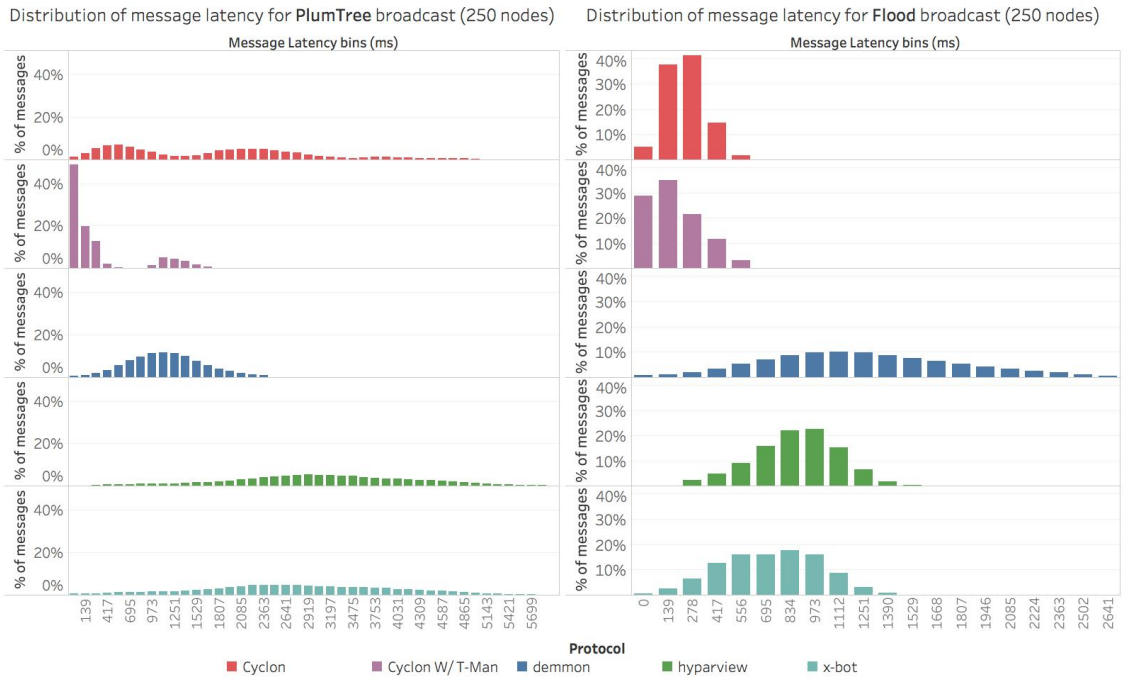

Figure 6.12: Message latency distribution in scenario with low network saturation

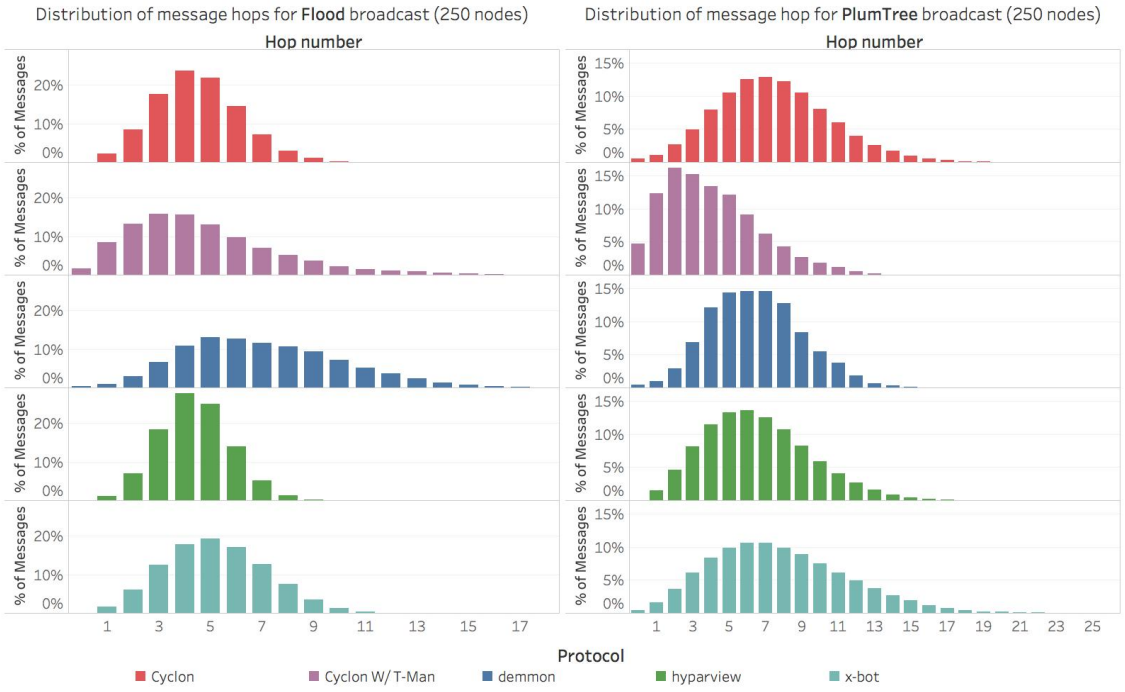

Figure 6.13: Message hop distribution in scenario with low network saturation

be explained by the fact that, by employing a single shared tree to disseminate the messages, as the messages must take specific routes in order to reach all nodes with decreased message redundancy, messages have to take more hops to get to their destination, and consequently achieve higher latency values. This behaviour is observable in Figure 6.13, which shows the hop distribution of the delivered messages in the same scenario of 250 nodes and on message emitted per node.

It is important to mention that, while Cyclon with T-Man achieves lower latency values in both tests, it does so at the cost of reliability, making it less applicable for a reliable broadcasting solution (as observed previously in the results displayed in fig. 6.6 and 6.7.

6.2.4 Summary

In this section, we covered the obtained results from the experimental evaluation of the devised membership protocol against multiple popular baseline protocols obtained from the study of the state-of-the-art. Two main aspects of the devised protocol were tested (at multiple scales): the first aspect was the ability to establish and maintain the overlay connections, where obtained results show that the devised protocol is consistently one of the fastest protocols to converge to a final topology. Furthermore, in regard to the latency values of the vertical connections of the established tree (excluding connections between nodes sharing the same parent in the tree, which are less used in general), DeMMon also achieves both the lowest average and total latency cost.

The second tested aspect of the devised protocol was their message dissemination capacity, where the devised protocol was evaluated against the previously mentioned benchmarks paired with two flood protocols: a simple flood and the PlumTree protocol. We conducted tests at both multiple scales and multiple failure rates and observed that while DeMMon tends to perform particularly well in regard to throughput at lower scales (50 nodes) when compared with any other tested protocol, while at larger node counts, its throughput tends to plateau at around the values as both X-Bot and Hyparview when paired with a simple flood. We also observed that, while tree topologies (both DeMMon and PlumTree) incur lower message redundancy, the use of a single shared tree for scenarios with multiple senders causes higher delays in messages when compared to simple flood alternatives, as messages take more hops to reach their destination.

To conclude, we believe the devised overlay performs competitively with popular state-of-the-art solutions for both creating and establishing an overlay network and for performing information dissemination. Conducted tests suggest that DeMMon performs better in saturation tests at lower node counts, indicating it as the most performant solution for these scenarios. However, for scenarios where message latency is a concern, results show that any tree approach (including DeMMon), although incurs lower networking costs, performs worse when compared to simple flood protocols.

6.3 Aggregation Protocol

In this section, we present and analyze the obtained results from the experimental evaluation of the devised decentralized aggregation protocol when compared with a popular monitoring solution from the state-of-the-art: named Prometheus [47]. We begin by providing the experimental setting and configuration settings used across the conducted experiments, then we present and discuss the obtained results from these experiments, and finish the Section by providing a summary along with the drawn conclusions from the evaluation of our solution in its monitoring and aggregation capacity.

The experimental setting in which the evaluation of our aggregation protocol was conducted on is the same as the one defined in 6.1, where each solution is tested using containers to multiplex the physical nodes, isolate the running processes, and apply both bandwidth capacity constraints and latency delays between nodes.

As previously mentioned in Section 4.3, the devised aggregation protocol offers three decentralized information collection primitives: neighbourhood, tree and global aggregation. In this section, we provide the obtained results regarding the evaluation of the tree and global aggregation features with comparable setups running Prometheus. Neighbourhood aggregation results are not shown as Prometheus does not provide a comparable feature (which we believe is already a result in itself). For all the conducted experiments, we tested the systems by collecting a certain aggregated value, calculated through the aggregation of a variable number of metrics, emitted at configurable intervals by dummy applications running in all the nodes of the system. The main criteria used to test the applicability of our solution was its error over time: obtained by comparing the aggregated value obtained by each node against their "supposed" value, according to the following formula:

$$Error(t) = \frac{|\sum localVal_i(t) - aggVal(t)|}{\sum localVal_i(t)}$$

Where $localVal_i$ corresponds to the emitted value of each node locally, $\sum localVal_i$ corresponds to the "real" value, and $aggVal$ corresponds to the obtained aggregated value during the experiment. In addition to the error over time, we collected other metrics to measure the performance of our solutions, such as the consumption of networking and computing resources. All tests were conducted with network sizes of 750 logical nodes, and for each experiment, we varied the number of metrics emitted by the dummy applications. Finally, for each of these experiment combinations, we conducted tests with failure rates of 0 and 50% of the nodes in the system, excluding the configurated tree roots.

The designed features were compared against Prometheus configured in two, distinct, tree-shaped setups: the first setup, which we named **centralized Prometheus**, corresponds to the most typical configuration of a Prometheus server, where a single server

collects and aggregates the metrics correspondent to all the nodes in the system. The second experimental setup, named **Prometheus tree**, corresponds to a more sophisticated setup where instead of having a single aggregating node, there is an intermediate layer of nodes aggregating the metric values of the leaf nodes. This intermediate layer, in turn, is aggregated by the root node (effectively splitting the load among the aggregator nodes). The root node, in this configuration, makes use of "federation" to scrape the partially aggregated value from the Prometheus servers in the intermediate layer.

In the following results, there are two parameters displayed for the Prometheus results, the first, denoted by i, corresponds to the size of the intermediate layer, the second parameter, denoted by o, corresponds to the number of servers to aggregate for each node in the intermediate layer (essentially, o corresponds to maxium number of leaf nodes in the promehtue). In addition, for both of the centralized and tree configurations, we also test setup a variation where every node in the system is an aggregator node (with the name **aggregator leaves**), which aggregates the metrics provided by their local dummy application and only export the aggregated value. It is important to mention that only the first two setups (centralized and tree) are, to our knowledge, the most representative of common Prometheus configurations, however, we include the aggregator leaves scenarios to study the impact in terms of network cost of performing in-transit aggregation by every node which emits metrics when compared to performing the aggregation process of metrics extracted from multiple nodes on a single node. An illustration of these setups can be found in Figure 6.14. In the following results, there are two parameters displayed for the Prometheus results, the first, denoted by i, corresponds to the size of the intermediate layer, the second parameter, denoted by o, corresponds to the number of servers to aggregate for each node in the intermediate layer (essentially, o corresponds to maxium number of leaf nodes per Prometheus server in the intermediate layer).

In addition, for both of the centralized and tree configurations, we also test setup a variation where every node in the system is an aggregator node (with the name **aggregator leaves**). In this setup, every node aggregates the metrics provided by their local dummy application, and only exports the aggregated value to the Prometheus server. It is important to mention that only the first two setups (centralized and tree) are, to our beliefs, the most representative of common Prometheus configurations. However, we include the aggregator leaves scenarios to study the impact in terms of network cost of performing in-transit aggregation by every node emitting metrics when compared to performing the aggregation process of metrics extracted from multiple in a centralized manner. An illustration of these setups can be found in Figure 6.14.

6.3.1 Tree aggregation

For the **tree aggregation** evaluation, we configured DeMMon with a single tree aggregation function, which triggers the algorithm defined in Section 4.3 that, in sum, collects an aggregated value of the metrics of its descendants in the DeMMon tree. This feature was

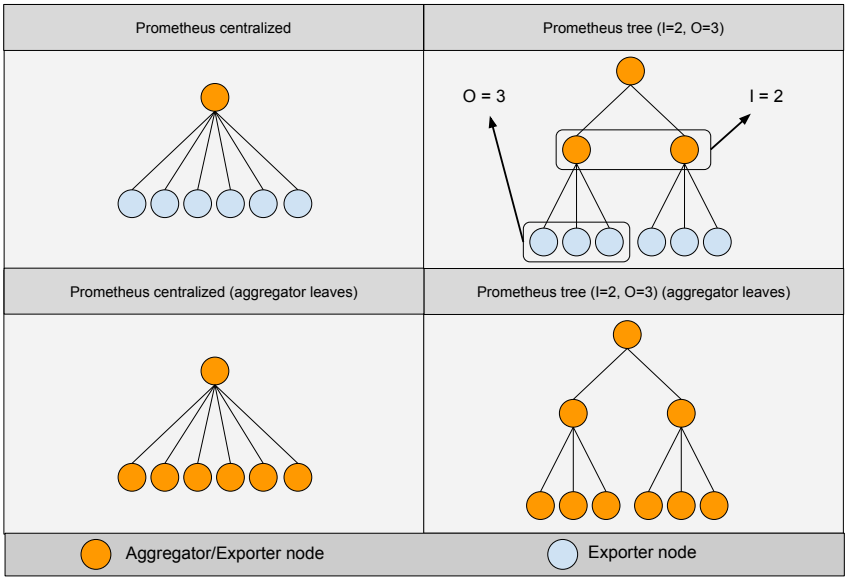

Figure 6.14: Exemplification (at smaller scale) the of tested prometheus setups

designed for decentralized resource management applications that follow the DeMMon hierarchical structure to perform decentralized resource management decisions. For example, a certain application that wishes to maintain a certain ratio of two service replicas (because one depends on the other), it can do so by having each node monitor its descendants and perform resource management actions (possibly coordinated with other nodes) to replenish or decommission a service replica such that the desired ratio is maintained.

To test this feature, we set up both Prometheus and DeMMon collecting an aggregated value of the whole system in a single node, providing an aggregated view of the system (our case, we used the sum to produce the aggregated value), and collected the error of the obtained aggregated value against the correct value over time. In addition to the error, we also collected the total network cost over the duration of the experiments.

We begin this comparison by discussing the obtained results regarding the centralized version of Prometheus against DeMMon (fig. 6.15), where we may observe that both DeMMon and Prometheus reach the 0% error values across all conducted tests, which means the systems are working correctly. Furthermore, we may observe that, in the regular Prometheus setup (non-aggregator leaves), as the number of metrics increases, Prometheus cannot obtain the metrics to calculate the aggregated value. We believe this occurs because the root node exceeds its allocated bandwidth. This contrasts with the Prometheus "aggregator leaves" results, which obtain 0% error value across all conducted experiments, which happens because every node is aggregating their emitted metrics and only propagating an aggregated value, which does not saturate the system bandwidth. It

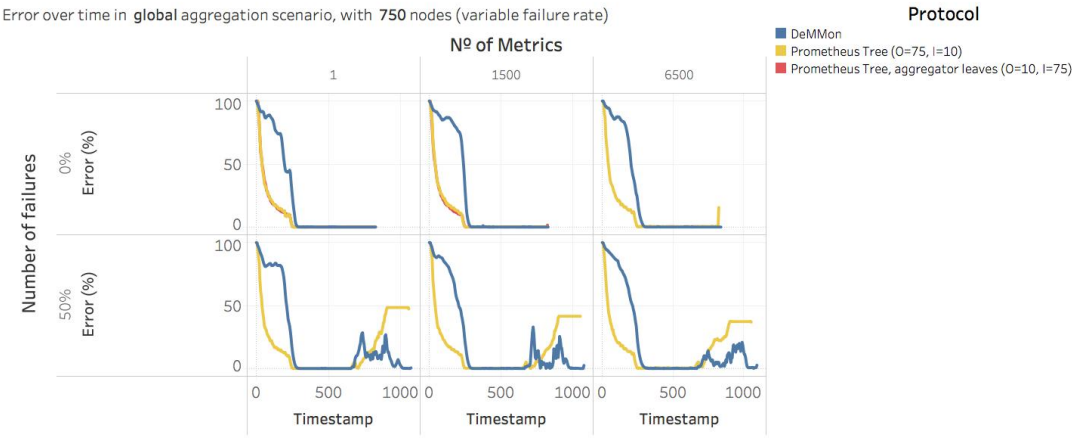

Figure 6.15: Error over time obtained in tree aggregation (centralized scenario)

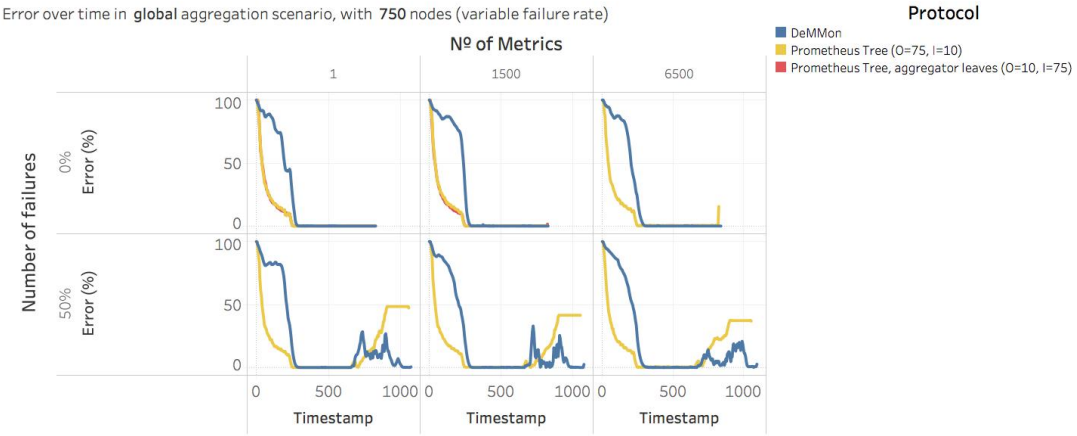

Figure 6.16: Error over time obtained in tree aggregation (tree scenario)

is important to notice that as the number of series increases, the DeMMon error tends to fluctuate between 0 and low error values. We believe this occurs due to the DeMMon nodes saturating their CPU when parsing the metrics. Although we realize this may be a limitation of the developed system, we argue this limitation is an engineering problem that may be easily addressed by employing a more efficient metric transmission and parsing protocol (similar to Prometheus' or InfluxDBs' [26]).

As the centralized Prometheus setup saturates at higher metric counts, we now compare the performance of our solution against a more scalable Prometheus setup, the **Prometheus tree** setup. The results of this comparison may be observed in Figure 6.16, which contains the error over time obtained for the experiments with 0 and 50% failure rates. As we can observe, Prometheus (in the non-aggregator leaves scenario) now splits the load of aggregating the metrics throughout multiple nodes and consequently can obtain the correct aggregated value. This configuration, however, is plagued by multiple

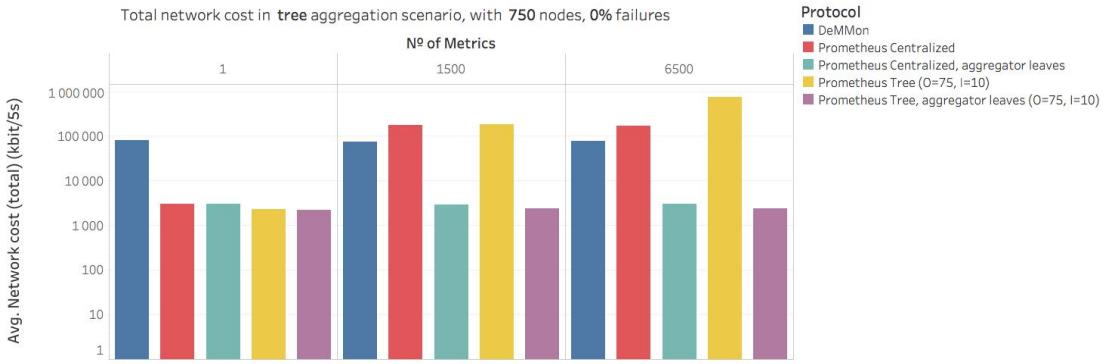

Figure 6.17: Average network cost incurred during tree aggregation experiments

unrecoverable points of failure, this is shown in the scenarios with 50% failures (bottom half of the graph), where Prometheus setups do not recover from the induced failures, as servers require manual intervention to change their configuration.

Finally, in Figure 6.17, we may observe the average network cost for the previously shown experiments. In this graph, we can observe that both the "aggregator leaves" incur a constant cost number, which is also the lowest obtained result when compared with other setups. This is expected because these setups perform aggregation of their metrics locally before emitting them towards the root node. In the case of DeMMon, the incurred networking cost is also constant, as DeMMon also performs local aggregation before emitting the results. In the case of what we believe to be the most representative Prometheus setups, the network costs tend to increase linearly with the number of metrics, which is less desirable when compared to a constant networking cost.

6.3.2 Global aggregation

Global aggregation in DeMMon, (as further explained in Section 4.3.3), is a feature where each node in the system calculates the result of the aggregation in a decentralized manner. To test the applicability of this feature, we also employed Prometheus as a baseline comparison, also configured with the setups described at the beginning of this section. However, for comparativeness, we configured each Prometheus server to node periodically query the root node for obtaining the aggregated value (effectively providing the same results as DeMMon).

The experiments conducted for this feature are similar, in terms of duration and failure rate, to the ones conducted for tree aggregation. Their results can be observed in Figures 6.18 and 6.19. In these, we observe a similar pattern to the one observed by the tree aggregation results, namely: the centralized Prometheus configurations cannot scale when the number of metrics emitted per node increases, and the tree Prometheus configurations, while they resolve the scaling problem, are subjected to multiple points of failure that, in the case of a failure, require manual configuration to recover.

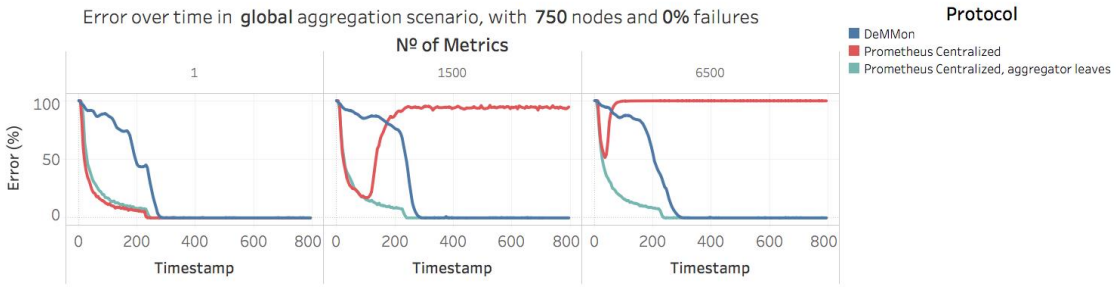

Figure 6.18: Error over time obtained in global aggregation (centralized scenario)

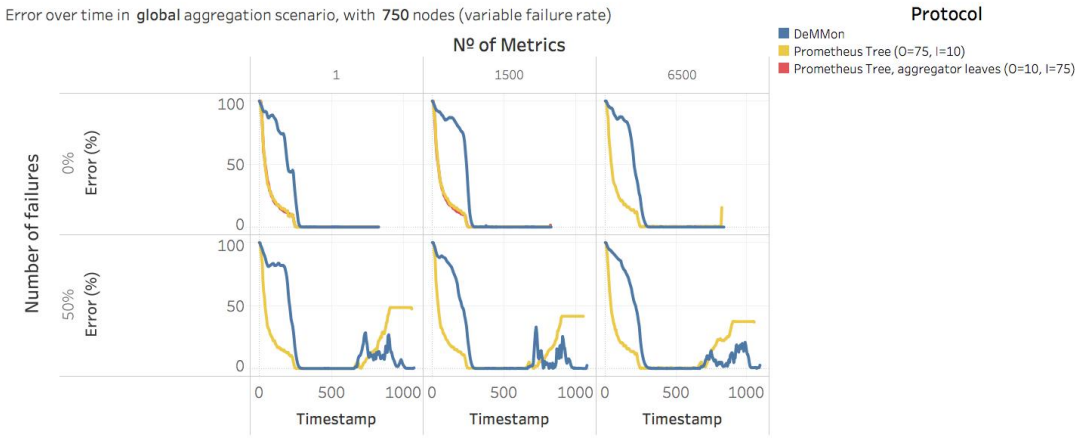

Figure 6.19: Error over time obtained in global aggregation (tree scenario)

Finally, in these results, we may observe that DeMMon can correctly obtain the aggregated value even when the number of metrics increases and in the presence of failures, making it a more versatile option for scenarios where failures are common, such as those found at the edge of the network.

The final obtained result, showing the average network cost incurred during these experiments, can be observed in Figure 6.20, where similarly to tree aggregation, both DeMMon and the two Prometheus configurations with "aggregator leaves" obtain constant network costs during the experiments (because these setups perform local aggregation before emitting their metrics). Furthermore, these Prometheus setups incur less networking costs, given they do not have to maintain the overlay network, unlike DeMMon.

The remaining configurations (Prometheus tree and Prometheus centralized), as they do not perform in-transit aggregation of the metrics, incur networking costs that scale with the number of metrics emitted by the nodes. Consequently, from the standpoint of scalability, this means its scalability will be limited by the number of series emitted per node, that contrary to DeMMon or configurations with "aggregator leaves" (which as previously mentioned, are not representative of most Prometheus setups), which obtain

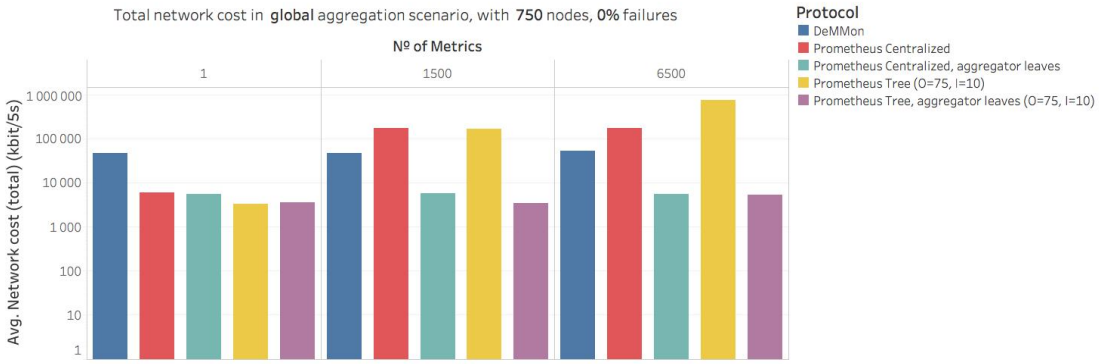

Figure 6.20: Average network cost incurred during global aggregation

linear networking costs with the number of emitted metrics.

6.4 Summary

In this chapter, we studied, through experimentation, the applicability of the devised decentralized aggregation and information dissemination framework, named DeMMon. We began by providing the system model in which we tested our solution (section 6.1), which aims to emulate a realistic cloud-edge scenario composed of nodes with heterogenous networking capacity and distributed among multiple places of the globe. Following, in Section

Finally, we concluded the chapter with Section 6.3, where we validated the implemented monitoring primitives and compared the obtained results against multiple configurations of a popular baseline solution: Prometheus. We showed that the devised decentralized monitoring solution obtains results comparable to those obtained by Prometheus, and that it provides a higher degree of fault-tolerance, as it does not require manual configuration to recover from failures.

7

Conlusions and future work

7.1 Conclusion

The edge computing paradigm has surged as the new up-and-coming solution to the limitations imposed by the cloud. The popularity of Edge Computing has grown as it allows applications to be enriched with new features, made possible by having available hardware closer to users. Furthermore, we believe this popularity is growing due to the size of the information produced outside the data centre, which following the current trend, will reach a point where the cloud infrastructure will start to become a bottleneck when performing timely and accurate management decisions.

For this reason, we argue in favour of performing decentralized resource management decisions, supported by partial knowledge of the system state, and enabled by devices within and outside the data centres. This thesis is focused on enabling this behaviour, through the development of a decentralized resource monitoring and management service, targeted for performing decentralized resource management actions in cloud-edge environments.

We studied not only the-state-of-the-art regarding these systems, but also the multiple areas which we believe to be crucial for building them: (1) the types of devices composing the cloud-edge environment, (2) how to federate these devices in self-optimizing overlay networks, and (3) how to perform in-transit aggregation using these overlay networks. Through this study, we built a Decentralized Management and Monitoring (DeMMon) service, which at its' root uses a devised overlay network protocol which employs the nodes' configured bandwidth, together with latency measurements to create and optimize a hierarchical tree-shaped network.

This overlay network was built using the first contribution of this thesis, named GO-Babel, which consists in an event-based framework designed to ease the building of distributed systems protocols. This framework is a port in Golang of Babel [1], however enriched with a fault detector and abstractions to perform latency measurements, which were a requirement for our protocol. Go-Babel's implementation has been validated through the conducted experimental work for the devised protocol and the baseline

protocols used in the overlay protocols' evaluation. This overlay was tested in realistic testbeds, focusing on its' capacity to establish and maintain the network even with node failures, and in its' capacity to propagate information (through broadcasting messages). The obtained results from the conducted experimental work not only show the protocols' validity, but also that the protocol can achieve higher throughput values when compared with state-of-the-art baselines in certain node counts, while remaining equal in throughput at other node counts.

Using this tree-shaped network, we implemented decentralized aggregation primitives that allow the collection of metrics across DeMMon nodes in three distinct manners: hierarchical aggregation (using the overlay tree), aggregation using a hop-based range, and global aggregation, which collects a globally aggregated value of the entire system. In our implementation, we took steps to ensure that the collection of metrics is possible to perform in an on-demand fashion and that whenever possible, nodes aggregating the same metrics reuse each others' values.

Similarly to before, we tested these features in a realistic testbed and compared their validity and applicability against a popular state of the art solution named Prometheus, which was configured with multiple tree-shaped setups. The obtained results show the validity of our solution, both with or without the presence of node failures, through achieving the correct aggregated values across multiple scenarios. Results also show that, as Prometheus configurations do not have to pay the price of maintaining the overlay network, these configurations incur lower networking costs when aggregating lower metric counts. However, as the number of metrics increases, results show that typical centralized Prometheus approaches cannot scale with as the numbers of metrics emitted increase, and Prometheus setups that do so by decentralizing the aggregation procedure become plagued with multiple points of failure which require manual configuration to recover.

The DeMMon framework provides access to these primitives through an API built (and a corresponding client) in WebSockets. In addition, this API allows resource management systems and other services, executing alongside DeMMon, to insert/retrieve metrics from a Time-Series database, and to request (potentially aggregated) metrics (through the previously mentioned decentralized aggregation primitives) from other nodes in the system, in which the DeMMon service is inserted into.

A goal of this work is to provide flexibility, consequently, aggregation functions which are employed to aggregate the metric values consist of user defined scripts. Furthermore, the metric values do not have a defined type (as long as they are serializable), which allows resource management applications to tailor these types to their needs, and to perform complex computations without resorting to a fixed set of aggregation functions, however at a cost of performance, when compared with better engineered state-of-the-art alternatives. In addition to this flexibility, the API also permits the execution of periodic functions to, for example, resample a time-series, or produce an aggregated value. The final presented API functionality is the ability for clients to install alarms. These allow clients to get notified whenever an issued condition is verified and prevent clients from

spending resources performing periodic verifications.

Finally, as the initial vision of this thesis was of pairing DeMMon with a decentralized service deployment system built by a colleague, we contributed to this vision through the collaborative creation of a benchmark edge-enabled application, intended to test the complete stack. This benchmark application aims to provide a decentralized implementation, through microservices, of an interactive multiplayer game with, functionality inspired from the popular game PokemonGO. We believe this contribution helps tackle what is currently an obstacle for the development of these systems, given that, to our knowledge, there were no free and open-source realistic implementations of edge-enabled interactive applications that we could use to test the complete envisioned software stack.

7.2 Future work

As future work, there are ideas prevenient from both the initial thesis vision and the development of our work. Regarding the overlay protocol, we believe there are multiple venues that may be taken, either to improve the existing protocol or to devise a new one based on the current presented work. First, we believe there are possible improvements to be had regarding reducing the necessary messages for performing maintenance of the established logical connections. This can be achieved through, for example, stopping the periodic emission of certain messages whenever the logical connections are established, consequently reducing the networking cost of maintaining the overlay. Second, we believe that it may be possible to insert additional mechanisms as an attempt to balance the overlay tree, aiming to decrease the number of hops and increase the latency incurred by messages sent in information dissemination scenarios. Finally, we believe there is the need to address the need for manual configuration and fault-recovery of the bootstrap nodes, this can be overcome either by creating a new specialized protocol to perform this function, or by embedding this feature into the protocol.

We also believe there is future work regarding the devised aggregation primitives, from adding new efficient aggregation primitives that rely on the abstractions provided by the tree to, similarly to the overlay protocol, reduce the overhead of maintaining the aggregation mechanisms. This can be realized, for example, by maintaining established aggregation trees by sending messages solely when nodes switch parent, nodes crash, or links fail by not periodically emitting messages to maintain the aggregation trees, instead of doing so with a periodic mechanism.

We believe another venue worth pursuing is to optimize the query evaluation process, namely experimenting with alternative libraries that allow the execution of user-defined code, or creating our own, tailored for our needs. In addition, we believe it would be valuable to replace the metric propagation protocol with a more efficient protocol (possibly not JSON-based).

Finally, as future work we plan to improve the stability of the built components by creating unit and integration tests, and to further separate the components through

interfaces which materialize this solution, such that each individual component can be replaced with another that satisfies the same interface. We believe our solution can be improved in this regard through, for example, separating the offered broadcast primitives from the membership protocol.

Bibliography

[1] URL: https://asc.di.fct.unl.pt/~jleitao/babel/ (cit. on pp. 3, 4, 29, 31, 99).

[2] URL: https://kafka.apache.org/ (cit. on p. 71).

[3] URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on p. 76).

[4] M. Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: https://doi.org/10.1145/1721654.1721672 (cit. on p. 1).

[5] D. Borthakur et al. "HDFS architecture guide". In: *Hadoop Apache Project* 53.1-13 (2008), p. 2 (cit. on p. 24).

[6] A. Brogi and S. Forti. "QoS-aware deployment of IoT applications through the fog". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1–8. ISSN: 23274662. DOI: 10.1109/JIOT.2017.2701408 (cit. on p. 26).

[7] A. Brogi, S. Forti, and A. Ibrahim. "How to best deploy your fog applications, probably". In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE. 2017, pp. 105–114 (cit. on p. 26).

[8] *Catch Pokemon in the Real World with Pokemon GO!* URL: https://www.pokemongo.com/en-us/ (cit. on pp. 26, 69).

[9] Y. Chawathe et al. "Making Gnutella-like P2P Systems Scalable". In: *Computer Communication Review* 33.4 (2003), pp. 407–418. ISSN: 01464833. DOI: 10.1145/863997.864000 (cit. on p. 17).

[10] B. Cohen. "Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. 2003, pp. 68–72 (cit. on p. 16).

[11] P. Costa and J. Leitao. "Practical Continuous Aggregation in Wireless Edge Environments". In: Oct. 2018, pp. 41–50. DOI: 10.1109/SRDS.2018.00015 (cit. on pp. 18, 20, 29, 31, 54, 55, 58).

[12] A. Crespo and H. Garcia-Molina. "Routing indices for peer-to-peer systems". In: *Proceedings 22nd International Conference on Distributed Computing Systems*. July 2002, pp. 23–32. DOI: 10.1109/ICDCS.2002.1022239 (cit. on p. 17).

[13] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 24).

[14] P. Druschel and A. Rowstron. "PAST: a large-scale, persistent peer-to-peer storage utility". In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. May 2001, pp. 75–80. DOI: 10.1109/HOTOS.2001.990064 (cit. on p. 13).

[15] *Empowering App Development for Developers*. URL: https://www.docker.com/ (cit. on p. 75).

[16] M. Finnegan. *Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic*. Mar. 2013. URL: https://www.computerworld.com/article/3417915/boeing-787s-to-create-half-a-terabyte-of-data-per-flight--says-virgin-atlantic.html (cit. on p. 1).

[17] P. Garbacki, D. H. Epema, and M. Van Steen. "Optimizing peer relationships in a super-peer network". In: *27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE. 2007, pp. 31–31 (cit. on p. 17).

[18] *Go is an open source programming language that makes it easy to build simple, reliable, and efficient software*. URL: https://golang.org/ (cit. on pp. 29, 32, 60).

[19] A. S. Grimshaw, W. A. Wulf, and C. The Legion Team. "The Legion Vision of a Worldwide Virtual Computer". In: *Commun. ACM* 40.1 (Jan. 1997), pp. 39–45. ISSN: 0001-0782. DOI: 10.1145/242857.242867. URL: https://doi.org/10.1145/242857.242867 (cit. on p. 1).

[20] *Gtk-Gnutella*. Dec. 2019. URL: https://sourceforge.net/projects/%20gtk-gnutella/ (cit. on p. 17).

[21] I. Gupta et al. "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 160–169 (cit. on p. 14).

[22] N. Hayashibara et al. "The /spl phi/ accrual failure detector". In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 2004, pp. 66–78. DOI: 10.1109/RELDIS.2004.1353004 (cit. on p. 31).

[23] B. Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22 (cit. on pp. 2, 24).

[24] C. H. Hong and B. Varghese. "Resource management in fog/Edge computing: A survey on architectures, infrastructure, and algorithms". In: *ACM Computing Surveys* 52.5 (2019). ISSN: 15577341. DOI: 10.1145/3326066 (cit. on pp. 23, 24).

[25] P. Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010 (cit. on p. 24).

[26] *InfluxDB data elements*. URL: https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/ (cit. on pp. 60, 94).

[27] *Internet Speed around the world*. URL: https://www.speedtest.net/global-index#mobile (cit. on p. 76).

[28] M. Jelasity and O. Babaoglu. "T-Man: Gossip-based overlay topology management". In: *International Workshop on Engineering Self-Organising Applications*. Springer. 2005, pp. 1–15 (cit. on pp. 11, 78).

[29] M. Jelasity, A. Montresor, and O. Babaoglu. "Gossip-Based Aggregation in Large Dynamic Networks". In: *ACM Transactions on Computer Systems* 23 (Aug. 2005), pp. 219–252. DOI: 10.1145/1082469.1082470 (cit. on p. 20).

[30] M. Jelasity et al. "Gossip-based peer sampling". In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es (cit. on p. 9).

[31] P. Jesus, C. Baquero, and P. S. Almeida. "A Survey of Distributed Data Aggregation Algorithms". In: *CoRR* abs/1110.0725 (2011). arXiv: 1110.0725. URL: http://arxiv.org/abs/1110.0725 (cit. on pp. 18, 19).

[32] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. June 2007, pp. 419–429. DOI: 10.1109/DSN.2007.56 (cit. on pp. 9, 12, 38, 78).

[33] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310 (cit. on pp. 10, 12, 83).

[34] J. Leitao, L. Rosa, and L. Rodrigues. "Large-scale peer-to-peer autonomic monitoring". In: *2008 IEEE Globecom Workshops*. IEEE. 2008, pp. 1–5 (cit. on p. 10).

[35] J. Leitão et al. "Towards Enabling Novel Edge-Enabled Applications". In: 732505 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989 (cit. on pp. 1, 6, 69).

[36] J. Leitão et al. "X-bot: A protocol for resilient optimization of unstructured overlay networks". In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2175–2188 (cit. on pp. 12, 78).

[37] J. C. A. Leitão and L. E. T. Rodrigues. "Overnesia: a resilient overlay network for virtual super-peers". In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE. 2014, pp. 281–290 (cit. on pp. 9, 12).

[38] C. Li et al. "Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management". In: *ACM Comput. Surv.* 51.2 (Apr. 2018). ISSN: 0360-0300. DOI: 10.1145/3154815. URL: https://doi.org/10.1145/3154815 (cit. on p. 1).

[39] J. Li et al. "Comparing the Performance of Distributed Hash Tables Under Churn". In: Mar. 2004. DOI: 10.1007/978-3-540-30183-7_9 (cit. on p. 13).

[40] J. Liang et al. "MON: On-Demand Overlays for Distributed System Management." In: *WORLDS*. Vol. 5. 2005, pp. 13–18 (cit. on p. 11).

[41] M. L. Massie, B. N. Chun, and D. E. Culler. "The ganglia distributed monitoring system: design, implementation, and experience". In: *Parallel Computing* 30.7 (2004), pp. 817–840 (cit. on p. 21).

[42] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65 (cit. on pp. 10, 14).

[43] Nm-Morais. *nm-morais/demmon-client*. URL: https://github.com/nm-morais/demmon-client (cit. on p. 65).

[44] Nm-Morais. *nm-morais/go-babel: Framework to build distributed systems protocols*. URL: https://github.com/nm-morais/go-babel (cit. on p. 29).

[45] *Network Testing Solutions*. URL: https://wondernetwork.com/ (cit. on p. 76).

[46] J. Paiva, J. Leitão, and L. Rodrigues. "Rollerchain: A DHT for Efficient Replication". In: *2013 IEEE 12th International Symposium on Network Computing and Applications*. Aug. 2013, pp. 17–24. DOI: 10.1109/NCA.2013.29 (cit. on pp. 10, 14).

[47] Prometheus. *From metrics to insight*. URL: https://prometheus.io/ (cit. on pp. 3, 22, 91).

[48] R. V. A. N. Renesse, K. P. Birman, and W. Vogels. "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining". In: *ACM Transactions on Computer Systems* 21.2 (2003), pp. 164–206 (cit. on pp. 10, 20–22).

[49] *rfc6455*. URL: https://datatracker.ietf.org/doc/html/rfc6455 (cit. on pp. 65, 67).

[50] Robertkrimen. *robertkrimen/otto: A JavaScript interpreter in Go (golang)*. URL: https://github.com/robertkrimen/otto (cit. on p. 62).

[51] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350 (cit. on pp. 13, 14, 21).

[52] A. Rowstron et al. "Scribe: The Design of a Large-Scale Event Notification Infrastructure". In: *Networked Group Communication*. Ed. by J. Crowcroft and M. Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 30–43. ISBN: 978-3-540-45546-2 (cit. on p. 13).

[53] *S2 Geometry*. URL: https://s2geometry.io/ (cit. on p. 72).

[54] M. Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (cit. on pp. 2, 25).

[55] *Self-driving Cars Will Create 2 Petabytes Of Data, What Are The Big Data Opportunities For The Car Industry?* URL: https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172 (cit. on p. 1).

[56] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. ISSN: 2372-2541. DOI: 10.1109/JIOT.2016.2579198 (cit. on p. 1).

[57] I. Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32 (cit. on pp. 10, 13).

[58] D. Stutzbach and R. Rejaie. "Understanding churn in peer-to-peer networks". In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM. 2006, pp. 189–202 (cit. on p. 8).

[59] *Swarm mode overview*. Aug. 2021. URL: https://docs.docker.com/engine/swarm/ (cit. on p. 75).

[60] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. "Theory and Practice of Bloom Filters for Distributed Systems". In: *IEEE Communications Surveys Tutorials* 14.1 (First 2012), pp. 131–155. ISSN: 2373-745X. DOI: 10.1109/SURV.2011.031611.00024 (cit. on p. 17).

[61] *The most popular database for modern apps*. URL: https://www.mongodb.com/ (cit. on p. 70).

[62] "Topology Management for Unstructured Overlay Networks." In: *Technical University of Lisbon* (2012) (cit. on pp. 8, 10, 15–17).

[63] V. K. Vavilapalli et al. "Apache Hadoop YARN: yet another resource negotiator". In: *SOCC '13*. 2013 (cit. on pp. 2, 24).

[64] A. Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17 (cit. on p. 24).

[65] N. Wang et al. "ENORM: A framework for edge node resource management". In: *IEEE transactions on services computing* (2017) (cit. on p. 25).

[66] P. Yalagandula and M. Dahlin. "A Scalable Distributed Information Management System". In: *SIGCOMM Comput. Commun. Rev.* 34.4 (Aug. 2004), pp. 379–390. ISSN: 0146-4833. DOI: 10.1145/1030194.1015509. URL: https://doi.org/10.1145/1030194.1015509 (cit. on pp. 21, 22).

[67] B. Zhao et al. “Tapestry: A Resilient Global-Scale Overlay for Service Deployment”. In: *IEEE Journal on Selected Areas in Communications* 22 (July 2003). DOI: 10.1109/JSAC.2003.818784 (cit. on p. 14).

Annex 1 - Extra figures

This annex is used to present the extra figures that were mentioned during the discussion of results in Section 6.2.3. These figures contain a summarized view of the obtained results regarding the conducted information dissemination experiments.

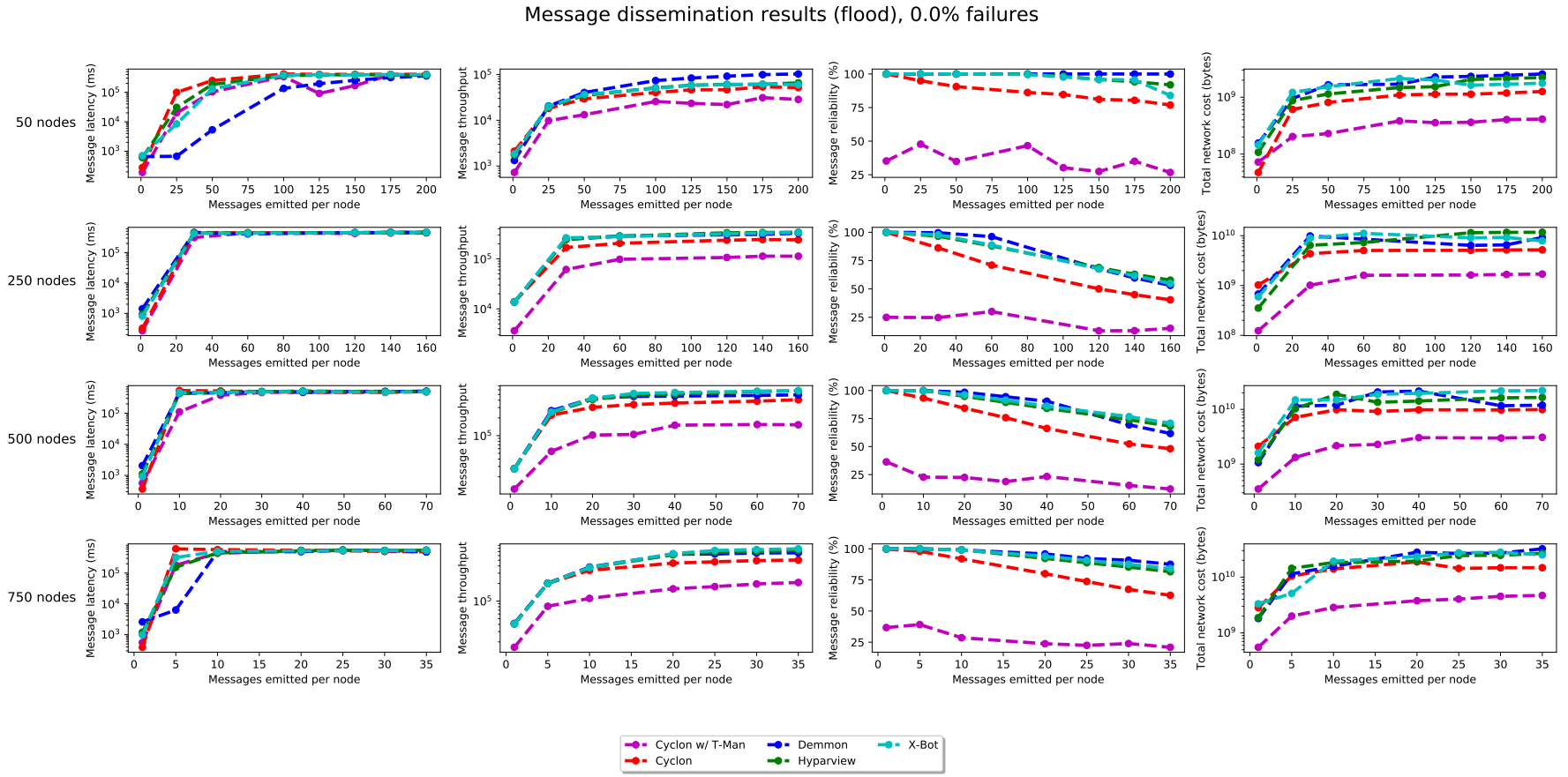

Figure I.1: Obtained results in simple flood scenario (0% failures)

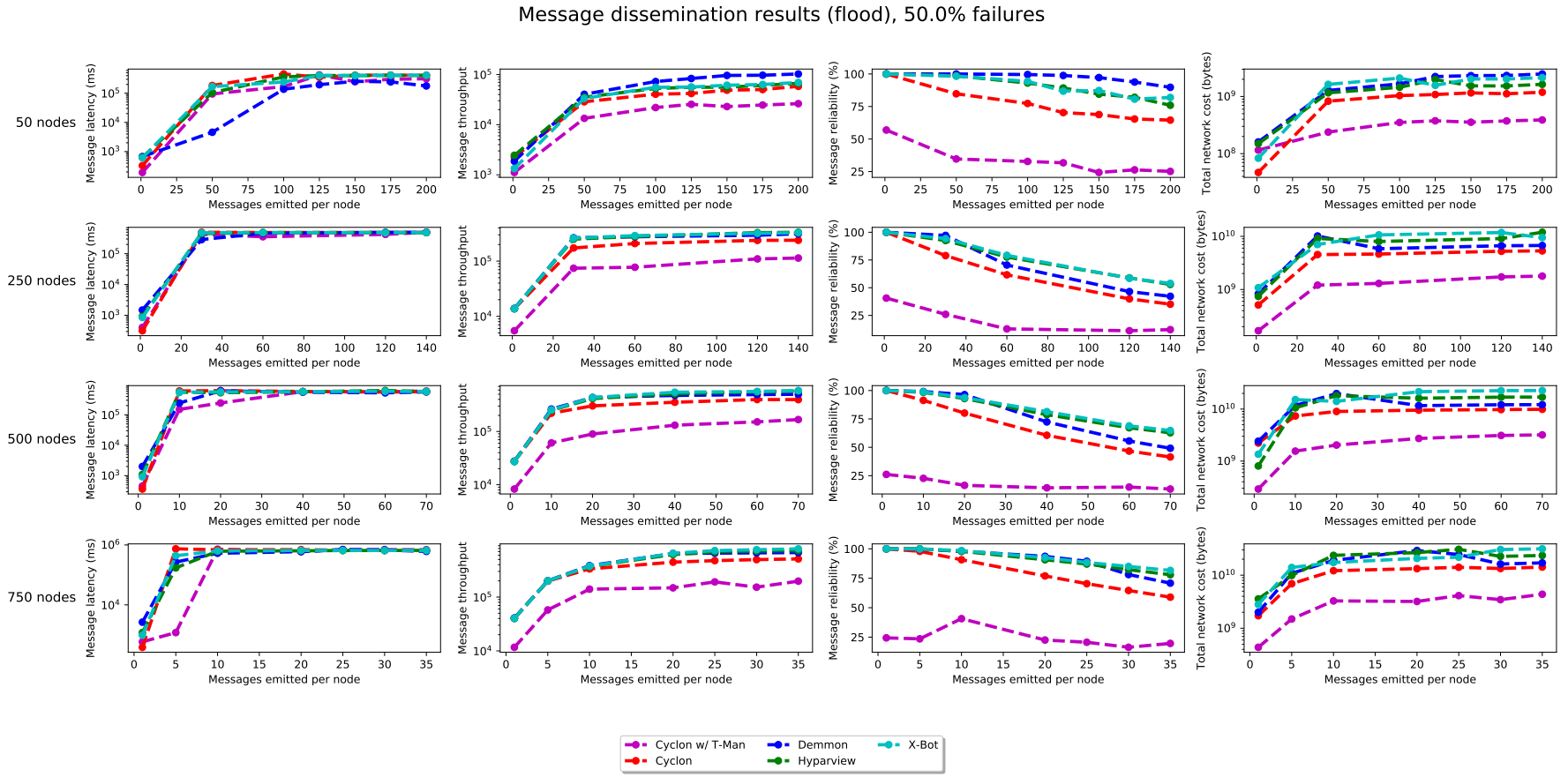

Figure I.2: Obtained results in simple flood scenario (50% failures)

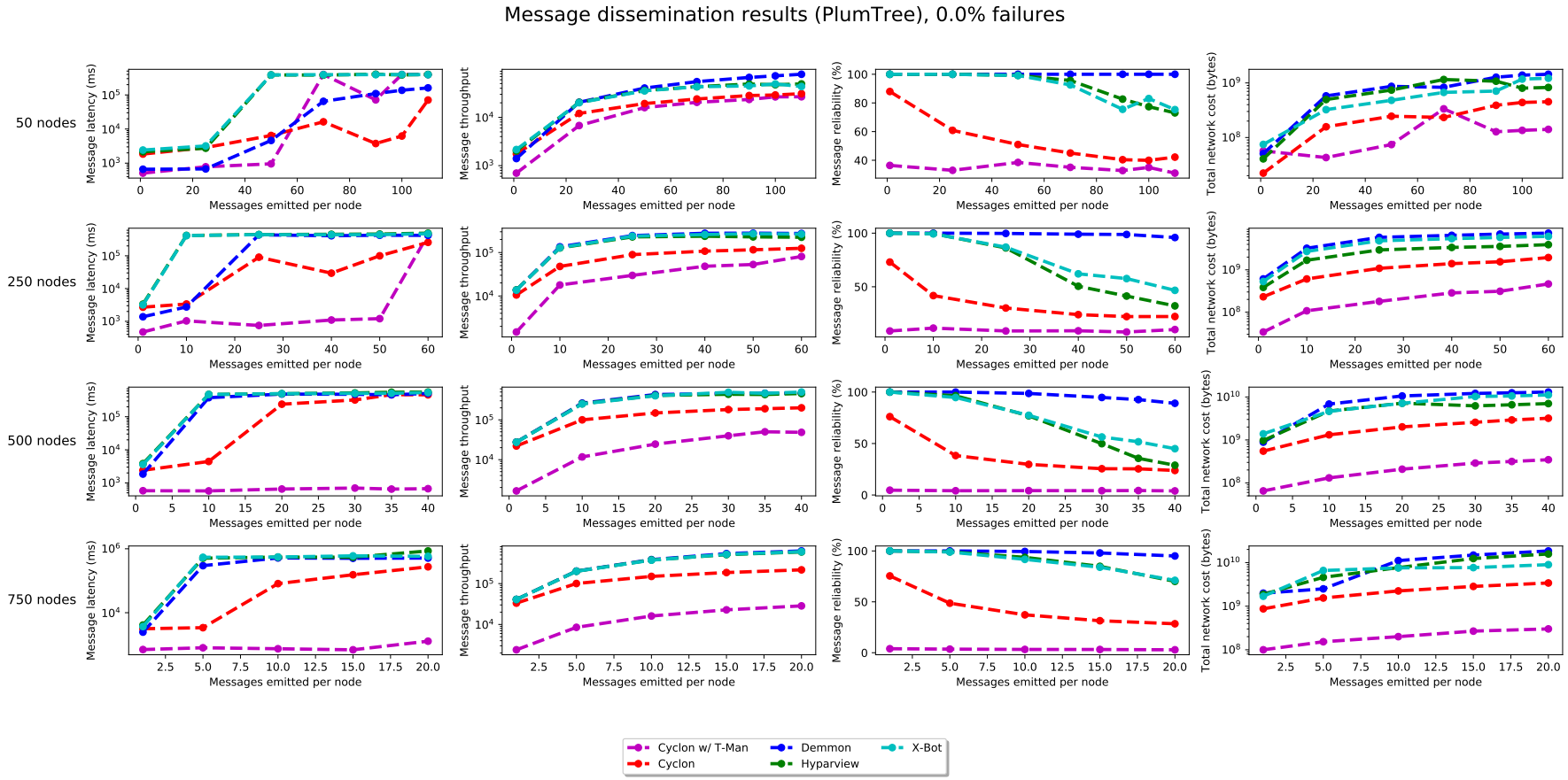

Figure I.3: Obtained results in PlumTree scenario (0% failures)

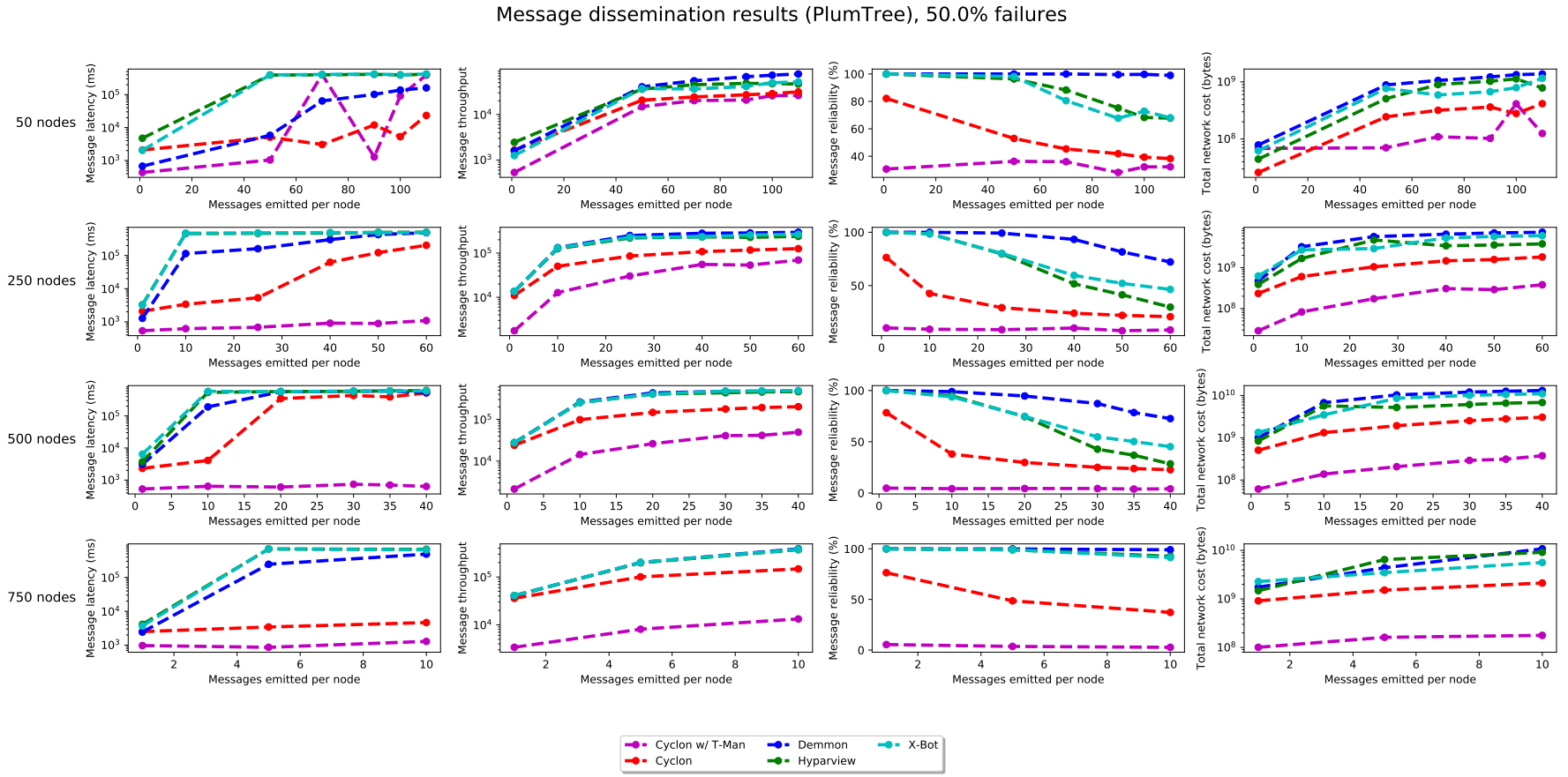

Figure I.4: Obtained results in PlumTree scenario (50% failures)