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 Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

⟨6⟩, ⟨2021⟩

DeMMonDecentralized management and monitoring framework

Dedicatory lorem ipsum.

Acknowledgements

Acknowledgments are personal text and should be a free expression of the author.

However, without any intention of conditioning the form or content of this text, I would like to add that it usually starts with academic thanks (instructors, etc.); then institutional thanks (Research Center, Department, Faculty, University, FCT / MEC scholarships, etc.) and, finally, the personal ones (friends, family, etc.).

But I insist that there are no fixed rules for this text, and it must, above all, express what the author feels.

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 data production and consumption is performed by devices outside of the data center. Serving data from and performing most of computations on cloud data centers increases the infrastructure costs for service providers and the latency for end users, while also raising security and privacy concerns.

The aforementioned limitations 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 data center as potential targets for computations. However, as edge devices tend to have heterogenous capacity and computational power, there is the need for them to effectively share resources and coordinate to accomplish tasks which would otherwise be impossible for a single edge device.

The study of the state of the art has revealed that existing resource tracking and sharing solutions are commonly tailored for homogenous devices deployed on a single stable environment, which are inadequate for dynamic edge environments. In this work, we propose to address these limitations by presenting a novel solution for resource tracking and sharing in edge settings. This solution aims to federate large numbers of devices and continuously collect and aggregate information regarding their operation, as well as the execution of deployed applicational components in a decentralized manner. This will allow edge-enabled applications, decomposed in components, to adapt to runtime environmental changes by either offloading tasks, replicating or migrating the aforementioned components.

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

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, impondo uma 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. O paradigma da Computação na Periferia propõe a execução de computações, e potencialmente armazenamento de dados, em dispositivos fora dos centros de dados, reduzindo custos e criando um novo leque de possibilidades para efetuar computações distribuídas mais próximas dos dispositivos que produzem e consomem os dados.

Neste artigo propõem-se o desenho de uma nova solução de monitorização e disseminação de informação descentralizada, desenhada para executar em sistemas de larga escala principalmente compostos por dispositivos com ligações de dados com capacidade limitada, como os que se encontram na periferia da rede. Esta solução baseia-se numa topologia descentralizada em árvore estabelecida entre dispositivos da periferia e da nuvem, que é utilizada para eficientemente disseminar, coletar, e processar informação relativa aos dispositivos (ou processos) em execução neste ambiente híbrido. A nossa solução foi avaliada em redes emulada de várias dimensões, chegando a um máximo de 750 nós, no contexto de disseminação e de monitorização. 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

Glossary

Acronyms

Symbols

1 Introduction

1.1 Motivation

Nowadays, the Cloud Computing paradigm is the standard for development, deployment, and management of services, most of the software systems present in our everyday life, such as Google Apps, Amazon, Twitter, among many others, are deployed on some form of cloud service. 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 [1]. It enabled the illusion of unlimited computing power, which revolutionized the way developers, companies, and users develop, maintain, and even use services.

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 [33]. All of these application domains are characterized by having data being generated and accessed (mostly) 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 exchange from the DC to the client as well as the latency in communication from the clients to the DC 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 [16] by taking into consideration computing and networking resources which exist beyond the boundaries of DCs, closer to the edge of systems [30] [48]. This paradigm 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 [47], and a Boeing 787 produces data at a rate close to 5 gigabytes per second [13], 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, we are faced with a huge increase in the number and diversity of computational devices, as these

range from Edge Data Centers to 5G towers and mobile devices. These devices, contrary to the cloud, have a wide range of computational capacity, and potentially limited and unreliable data lines. Given this, developing an efficient resource monitoring and management platform which enables the adequate and efficient use of these devices is an open challenge for fully realizing in Edge Computing.

improve

1.2 Context

Resource management platforms are extensibly used in Cloud systems (e.g. Mesos [19], Yarn [54], Omega [46], among others), whose high-level functionality consist 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) perform ving 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 are being met.

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 global knowledge of the system. Although this system architecture heavily simplifies the management of the resources, we argue that such systems are plagued by a central point of failure and a single point of contention that hinders the scalability of such solutions, making them unsuitable for the scale of Edge Computing systems.

Instead, for achieving general-purpose computation in Edge systems, we argue in favour of decentralized management and monitoring platforms, composed of multiple components, organized in a flexible hierarchical way, and promoting load management decisions supported by partial and localized knowledge of the system. As building such a platform would not be trivial, and as we believe that in such a system, the accuracy and freshness of the information (which may be but is not exclusive to the execution of components or services) each component has, dictates how efficiently they manage resources. As such, we focus on that particular task: information gathering and aggregation.

We believe data aggregation is an essential step towards general-purpose computations in Edge systems, as it allows information to be summarized. For devices with constrained data links and limited resources in resource management systems, being able to summarize data in transit is crucial, as it provides them with a partial view of the aggregated value, which in turn can be used in decentralized resource management decisions (e.g. load-balancing, improving QOS, among others).

1.3 Contributions

The contributions which arose from this dissertation are as follows:

1. 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 overlay network using both bandwidth and node latency as heuristics for connection establishment. This protocol is fully decentralized and fault-tolerant, with its only configuration being a set of static nodes: the roots of the trees.

   b) A distribured aggregation protocol, which uses the connections created by the overlay protocol's tree structure to perform efficient on-demand in-transit aggregations.

   c) An API to consult information regarding the operation of the overlay protocol, and to issue or collect arbitrary information in the framework in the form of time-series data. This framework also allows other auxiliary functions such as applying periodic functions to information, or alerting based on provided information.

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

   e) An experimental evaluation of the monitoring protocol against common prometheus configurations. Here, the accuracy of the collected monitoring values is collected over time, as well as information regarding the networking/processing cost of collecting the information.

2. A benchmark in the form of an edge-enabled application composed by multiple loosely coupled micro-services, tailored to benchmark resource management platforms, in this benchmark, geographical proximity leads to a significant improvement of QOS for the end-user, favouring resource management platforms which value placement of their services closer to the client.

1.4 Document structure

elaborate this

The remaining of this document is structured as follows:

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

2

Related Work

The goal of this chapter is to present the related work studied that is associated with our objectives. We begin by identifying the four high-level requirements of a 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 application component or service.

3. *Resource Monitoring* studies which metrics to track per device, and how to efficiently compress those metrics through aggregation to reduce the size of the collected data, as well as how to propagate that data towards the components that need it to operate.

4. *Resource Management* addresses how to efficiently manage system resources and schedule jobs across existing resources such that: (1) the system remains load-balanced; (2) operations can operate efficiently; (3) jobs have data locality; and (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, we also discuss this aspect here.

Considering the identified high-level components of such a system, in the following sections we cover the taxonomy of devices which compose the edge environment, and discuss how they can be employed towards the design of the proposed solution (Section 2.1). Next, we study execution environments for applications, namely virtual machines

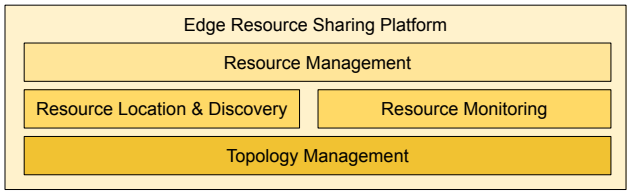

Figure 2.1: High-level architecture for a resource sharing platform

and containers, discuss their performance impact as well as their strengths and limitations towards supporting edge-enabled applications (Section 2.2).

Following, we study how to federate devices in an efficient abstraction layer that establishes an efficient topology (Section 2.3), and address how peers can efficiently index and search for the resources they need (e.g. services, peers, computing power, among others) in the aforementioned abstraction layer, which in turn enables the delegation of particular application components (Section 2.4). This is important given the fact that edge devices are typically resource constrained, and a computing task which would otherwise require a single cloud device, may require multiple edge devices to be accomplished in an efficient way.

Next, we cover tools to collect metrics from the aforementioned execution environments that are relevant towards performing efficient resource allocations. We analyze how to aggregate those metrics in a decentralized manner, and discuss relevant resource monitoring systems in the literature, for each, we address its limitations and advantages for the edge environment (Section 2.5). Lastly, we cover the taxonomy of resource management solutions, and present popular systems in the literature that share aspects with the solution we aim at developing (Section **??**).

2.1 Edge Environment

In this section we provide context about edge-related paradigms, study the taxonomy of the devices which materialize edge environments, and analyze which computations each device can perform.

2.1.1 Edge Computing

As previously mentioned, edge computing calls for the processing of data (and potentially storage) across all the devices which act as an "edge" along the path from the data center (DC) do the data source or client device [30]. It has the potential of enabling novel edge-enabled applications along with optimizing existing systems [48], making them more responsive.

Many approaches have already leveraged on some form of Edge computing in the past. **Cloudlets** [55] are an extension of the cloud computing paradigm beyond the DC, and consist in deploying resource rich computers near the vicinity of users that provide cloud

functionality. They have become a trending subject and have been employed towards resource management, Big Data analytics, security, among others. A limitation of Cloudlets is that because they are specialized computers, they cannot guarantee low-latency ubiquitous service provision, and cannot ensure that applications behave correctly in the presence of large hotspots of users.

Content Distribution networks [40] are specialized high bandwidth servers strategically located at the edge of the network which replicate content from a certain origin and serve it at reduced latencies, effectively decentralizing the content delivery.

Fog Computing [3] is paradigm which aims at solving similar problems to the Edge Computing. It proposes to provide computing, storage and networking services between end devices and traditional cloud DCs, typically, but not exclusively located at the edge of the network. We consider Fog Computing to be interchangeable with our definition of Edge Computing, however, with a special emphasis on providing infrastructure for edge-enabled services, instead of focusing on the inter-cooperation among devices.

Osmotic Computing [56] envisions the automatic deployment and management of inter-connected microservices deployed over a seamless infrastructure composed of both edge and cloud devices. This is accomplished by employing an orchestration technique similar to the process of "osmosis". Translated, this consists in dynamically detecting and resolving resource contention via the execution of coordinated microservice deployments / migrations across edge and cloud devices. This paradigm is a subset of Edge Computing, as it only focuses on deploying microservices on edge devices instead of employing them towards generic computations, in addition, the original authors only envision deploying services over cloud and edge DCs, instead of the whole range of possible devices.

Multi-access edge computing [36] (MEC) is a network architecture which proposes to provide fast-interactive responses for mobile applications. It solves this by employing the devices in the edge (e.g. base stations and access points) to provide compute resources for latency-critical mobile applications (e.g. facial recognition). Similar to Osmotic Computing, we consider MEC a subset of edge computing, given that its primary focus is on how to offload the computation from mobile to the cloud and not vice-versa.

2.1.2 Edge Environment Taxonomy

According to Leitão et al. [30], 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.

2.1.3 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 and have lower computational power. Consequently, the higher the level, the harder it is to employ edge devices to support the execution of edge-enabled applications.

Devices in levels 0-5 are potential candidates towards building the resource management and monitoring system we intend to create. 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. This effect can be circumvented by employing devices in other levels as gateways for those devices, hence starting to establish a hierarchy on the way different application components interact.

2.2 Execution Environments

After studying the taxonomy of the edge environment, it is paramount to study how these devices can execute computations (e.g. hosting application components, monitoring tasks, among others) in a controlled environment. A major requirement of these environments is the ability to simultaneously execute multiple computations, and that these interfere as little as possible with each other, as well as with the core behavior of the system.

A popular approach towards solving these challenges is to perform computations in loosely coupled independent components running some form of virtualization software, as it enables the co-deployment of components within the same physical machine. The main benefits of employing virtualization include hardware independence, isolation, secure user environments, and increased scalability.

The two most common types of virtualization used nowadays are containers and virtual machines (VMs), in this section present a brief description of both technologies, and study their advantages / limitations towards supporting edge-enabled applications.

2.2.1 Virtual Machines

A VM provides a complete environment in which an operating system and many processes, possibly belonging to multiple users, can coexist. By using VMs, a single-host hardware platform can support multiple, isolated guest operating system environments simultaneously [49].

Virtual machines rely on a type of software called a *hypervisor*, the role of the hypervisor is to abstract hardware to support the concurrent execution of full-fledged operating systems (e.g. Linux or Windows). Virtualizing the hardware layer ensures great isolation between virtual machines, meaning that a VM cannot directly interact with the host or the other VMs, which is highly desirable for both the virtualized applications and the host.

However, virtualizing the hardware and the device drivers incurs non-negligible overhead, and the large image sizes of operating systems required by virtual machines makes live migrations harder to accomplish, which we believe to be crucial in edge environments.

2.2.2 Containers

Containers (e.g., Docker [12], Linux Containers [20], among others) can be considered as a lightweight alternative to hypervisor-based virtualization. When using containers, applications share an OS (and maybe binaries and libraries), and implement isolation of processes at the operating system level. As a result, these deployments are significantly smaller in size than hypervisor deployments, for comparison, a physical machine may store hundreds of containers versus a few tens of VMs [2].

In terms of performance, container-based virtualization can be compared to an OS running on bare-metal in terms of memory, CPU, and disk usage [41], and contrary to VMS, restarting a container doesn't require rebooting the OS [2], meaning that a small-sized computation task may be accomplished much faster.

Consequently, given their lightweight nature, it is possible to deploy container-based applications (e.g. microservices), which can perform fast migration across nodes in the edge environment (e.g. in order to improve quality of service (QoS) of applications). This flexibility towards the migration process is an effective tool to deal with many challenges such as load balancing, scaling, resource reallocation and fault tolerance.

2.2.3 Discussion

Although VMs are widely present in the cloud infrastructure, they incur significant start up time (due to having to start-up an entire OS) and image sizes are larger when compared to containers (due to requiring a full OS image), which hinders the ability to perform quick migrations across different devices. The accumulation of these factors make VMs unsuited for devices with low capacity and availability, which are abundant in edge environments, consequently, we believe containers are the most appropriate solution when it comes to performing resource sharing in edge scenarios.

2.3 Topology Management

A major challenge towards decentralized resource monitoring and control, is to federate all devices (that we also refer to as peers following the peer-to-peer (P2P) literature) in an abstraction layer (an overlay network) that allows intercommunication and efficient resource discovery. This section provides context regarding the taxonomy of overlay networks, followed by a discussion of popular overlay network protocols.

In a P2P system, peers contribute to the system with a portion of their resources, so that the overall system can accomplish tasks which would otherwise be impossible for a single peer to solve. Typically, this is achieved in a decentralized way, which means peers must establish neighboring connections among themselves to enable information exchange which, in turn, enables to 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, this 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 [51]), this information becomes costly to maintain up-to-date.

In order to circumvent the aforementioned problems, 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 which restricts neighboring relations among peers. Partial membership solutions are attractive

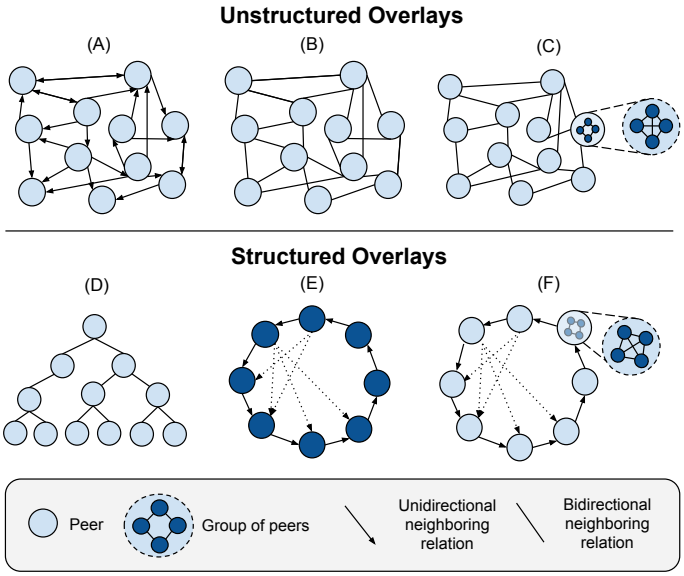

Figure 2.2: Examples Overlay Networks

because they offer similar functionality to full membership systems, while achieving more scalability and resiliency to churn. The closure of these neighboring relations is what materializes an **overlay network**.

2.3.1 Taxonomy of Overlay Networks

Overlay networks are logical networks which operate at the applicational level, these rely on an existing network (commonly referred to as the *underlay*) to establish neighboring relations, where each participant tipically only communicates directly with its overlay neighbors [53]. Overlays are commonly designed towards specific applicational needs, as such, their neighboring relations may or may not follow some sort of logic. As observable 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 neighboring relations, peers may pick random peers to be their neighbors, or alternatively employ strategies to rank neighbors and selectively pick the best given a particular criteria, that is typically entwined with the needs of applications. A key factor of unstructured overlays is their low maintenance cost, given that nodes can easily create neighboring 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 ilustrate three examples of unstructured overlay networks: (A) is a representation of an overlay network where the connections are unidirectional (e.g. Cyclon [24]), 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, neighboring 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 [27] 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 [32]). Grouping multiple devices into a group can be useful because: (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) for devices with low computing capabilities, groups are useful because nodes have nearby neighbors which can simplify the offload of computational tasks.

Structured Overlays

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

In Figure 2.2 illustrate three kinds of structured overlay networks: (D) corresponds to a tree, trees are widely used to perform broadcasts (e.g., PlumTree [28]) 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 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 [43])). However, trees are very fragile in the presence of faults [28].

Overlay depicted in (E) corresponds to the overlay topology tipically expected to support Distributed Hash Tables. These overlays are extremely popular due to their effective applicational-level routing capabilities. In a DHT, peers employ a global coordination mechanism which restricts their neighboring relations such that can find any peer *responsible* for any given key in a small limited number of steps.

In the example that we show in (E), the topology consists of a ring (which is the strategy employed by Chord [50]), however, not all distributed hash tables rely on rings to perform effective routing. For example, in Kademlia [38], nodes organized as leaves across a binary tree.

Finally, the overlay denoted in (C) is similar to overlay (E), however, each position of the DHT consists of a virtual node composed by multiple physical nodes (which is the strategy employed by Rollerchain [39]). Because of this, routing procedures have the potential to be load-balanced, and churn effects are mitigated, because the failure of a physical node does necessarily mean the failure of a virtual node.

2.3.2 Overlay Network 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 [53]:

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 neighboring relations across the neighbors of links between a given node. It consists in the number of a node's neighbors divided by the maximum number of links that could exist between those neighbors. 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.

2.3.3 Examples of Overlay Networks

T-MAN [22] is 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, etc.). To achieve this, T-MAN expects a topology as an input to the protocol, this topology is then materialized by employing a ranking method which is applied by every node to compare the preference among possible neighbors iteratively.

Nodes periodically exchange their neighboring 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 stability of the in-degree of nodes during the optimization of the overlay, and consequently, the overlay may not remain connected.

Management Overlay Network [35] (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, consequently, MON has a very low maintenance cost when there are no commands running.

The on-demand overlay construction allows the creation 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 [27] (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, it is maintained by a simple gossip protocol which periodically sends a message to a random peer in the active view. This message contains a subset of the neighbors of the sending node and a time-to-live (TTL), the message is forwarded in the system until the TTL expires, updating the views of nodes it is forwarded to. In contrast, the *active view* consists in a smaller view (around log(n)) created during the bootstrap of the protocol, and actively maintained by monitoring peers with a TCP connection (effectively making the active view connections bidirectional and act as a failure detector). Whenever peers from the active view fail (detected by the active TCP conntection), nodes attempt to replace them with nodes contained 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 [28]). It achieves high reliability even in the face of high percentage of node failures, however, the resulting topology is flat, which is not desirable given the taxonomy of edge environments we are considering. Furthermore, it may suffer from topology mismatch, because of the random nature of neighboring connections, the resulting neighboring connections may be very distant in the underlying network.

X-BOT [31] is a protocol which constructs an unstructured overlay network where neighboring relations are biased considering a particular, and parametrizable, metric. This metric is provided by an *oracle*, the oracle is a component that exports a function which accepts a pair of peers and attributes a cost to that neighboring connection, this cost may take into account 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 [27]. Then, nodes periodically trigger optimization rounds where they attempt to bias a portion of their connections according to the oracle. This potentually

addresses the previous concerns about the overlay topology mismatching the underlying network, however, it still proposes a flat topology, which is also not adequate for the edge environment taxonomy.

Overnesia [32] is a protocol which 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 request to a bootstrap node which triggers 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).

Intra-group membership consistency is enforced by an anti-entropy mechanism where nodes within a group periodically exchange messages containing their own view of the group. When a group detects that its size has become too large, it triggers a dividing procedure where splits the groups in two halves. Conversely, when the group size has fallen bellow a certain threshold, nodes trigger a collapse procedure, where 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 enables load-balancing, efficient dissemination of queries, and fault-tolerance. However, limitations from Overnesia arise from peers maintaining active connections to all members belonging to the same group, and keeping the group membership up-to-date, which may limit system scalability, finally, the overlay may suffer from topology mismatch, as two nodes within the same group may be distant in the underlay.

Chord [50] 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 that is uniformly distributed in the id space. Then, peers are ordered by identifier in a clockwise ring, where any data piece identified by k, is assigned to the first peer whose identifier is equal or follows k in the identifier space.

Chord implements a system of "shortcuts" called the *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 the a good trade-off between bandwidth and lookup latency [34], 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. 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 [44] is another well known DHT which assigns a 128-bit node identifier (nodeId) to each peer in the system. The nodes are randomly generated, and consequently, are uniformly distributed in the 128-bit nodeId space. Routing procedures are as follows: in each routing step, messages 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 [45] and file storage systems like PAST [11]. 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 node ID values, and complicates attempts at formal analysis of worst case behavior, in addition to establishing a flat topology which mismatches the edge device taxonomy.

Tapestry [58] Is a DHT similar to Pastry [44], however, nodeIDs are represented taking into account a certain base b supplied as a parameter of the system. In routing procedures, messages are incrementally forwarded to the destination digit by digit (e.g. ***8 -> **98 -> *598 -> 4598), consequently, routing procedures theoretically take logb(n) hops to 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 entries at each route level, consequently, nodes contain entries for a fixed-sized neighbor map of size b.log(N).

Kademlia [38] 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 [44]) 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 [18] 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 [39] 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 maintenance cost when compared to a regular DHT.

2.3.4 Discussion

Unstructured overlays are an attractive option towards 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 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 neighboring rules, participating nodes cannot replace neighbors 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.

2.4 Resource Location and Discovery

Resource location systems are one of the most common applications of the P2P paradigm [53], 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 achieve this, a search strategy must be applied, which depends on both the structure of an overlay network (structured or unstructured), on the characteristics of the resources, and on the desired results. For example, in the context of resource management, if a peer wishes to offload a certain computation to other peers, one 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 by the studying 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.

2.4.1 Querying techniques

Querying techniques consist of how peers describe the resources they need. Following, we cover common querying techniques employed in resource location systems [53]: **(1) Exact Match queries** specify the resource to search by the value of an unique attribute (i.e., an identifier, commonly the hash of the value of the resource); **(2) keyword queries** employ one or more keywords (or tags) combined with logical operators to describe resources (e.g. "pop", "rock", "pop and rock"...); **(3) range queries** retrieve all resources whose value is contained in a given interval (e.g. "movies with 100 to 300 minutes of duration"); **(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 resources with 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**.

2.4.2 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 [6].

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, which makes them unsuited for edge environments.

2.4.3 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, which is accomplished by disseminating queries through the overlay, there two popular approaches for accomplishing this in unstructured overlays: **flooding** and **random walks** [53].

Flooding consits in peers eagerly forwarding queries to other peers in the system as soon as they receive them for the first time, the objective of flooding is to contact a certain number of 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 TTL to messages that limits the number of times a message 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 which are more likely to have answers [8], this technique called a **random guided walk**. A common approach to bias random walks is to use bloom filters [52], 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 [17]) 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 [4], employed towards effectively reducing the number of peers that have to disseminate queries on the second version of Gnutella [17].

SOSP-Net [14] (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.

2.4.4 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) which are 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) [53].

One particular type of DHT that is commonly employed in small sized resource location systems is the One-Hop Distributed Hash Table (DHT), nodes in a one-hop DHT have full membership of the system and, consequently, they can locally map resources to known peers and perform lookups in O(1) time and message complexity. Facebook's Cassandra [26] and Amazon's Dynamo [9] are widely used implementations of one-hop DHTs.

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 system.

The second technique consists in transferring the entire resource to the responsible node in the DHT, contrasting to the previous technique, the resources are not replicated: due to consistent hashing, all nodes with the same resource will publish the resource in the same location of the DHT.

2.4.5 Discussion

As mentioned previously, centralized resource location systems are unsuited for edge environments, given that devices have low computational power and storage capabilities, it is impossible for an edge device to index all the resources in a system.

Unstructured resource location systems are attractive to perform queries that search for resources which are abundant in the system, 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 especially tailored towards exact match queries, but are less robust to churn and are subject to low-capacity nodes being a bottleneck in routing procedures.

In the context of the proposed solution, given that the resources we intend to manage are present in all nodes (e.g., computing power, memory, among others), we believe unstructured resource location is more suited. For example, if an edge device wishes to find nearby computing resources to offload a certain task, it may employ a random walk. On the other hand, if a peer wishes to find a larger set of computing resources to deploy multiple application components, it may employ flooding techniques.

2.5 Resource Monitoring

In this section we will cover **resource monitoring**, which consists in tracking the state of a certain aspects of a system, such as the device status, the capacity of links between devices, the status of available resources in a given zone of the system, among others.

Resource monitoring is paramount for making effective management decisions regarding task allocations and managing the overlay network.

2.5.1 Device Monitoring

A particularly hard problem in resource monitoring is fault detection, given the need to ensure each component is monitored by at least one non-faulty component, even in the face of joins, leaves, and failures of both nodes as well as network infrastructure. Most fault-detectors rely on heartbeats, which consist in a peer sending a message periodically to another peer in order to signal that it is functioning correctly.

Leitao, Rosa, and Rodrigues [29] proposes a decentralized device monitoring system by employing Hyparview [27] as a decentralized monitoring fault detector, given the fixed number of active connections, which ensures overlay connectivity, each peer will have at least another non-faulty component monitoring it through the active TCP connection.

In addition to tracking device health, it is paramount to collect metrics regarding the operation of the device, such as: **(1) Network related metrics**: devices need to be interconnected across an underlying infrastructure which is continuously changing. This raises concerns about the network link quality between devices across the system, especially if they are running time-critical services. Given this, it is paramount to track network related metrics such as bandwidth, latency and link status. **(2) Memory related metrics:** either related to volatile memory or persistent memory, it is important to track the amount of free and used memory. **(3) CPU metrics**: the utilization of the CPU (e.g., user, sys, idle, wait).

2.5.2 Container Monitoring

As previously mentioned, containers are the solution which incurs less overhead when it comes to sharing resources in the same node, given this, we now study tools which monitor the status of containers and the applications executing inside them.

Docker [12] has a built tool called **Docker Stats** [10] which provides a live data stream of metrics related to running containers. It provides information about the network I/O, cpu and memory usage, among others.

Container Advisor [15] (cAdvisor) is a service which analyzes and exposes both resource usage and performance data from running containers. The information it collects consists of resource isolation parameters, historical resource usage and network statistics. cAdvisor includes native support for Docker containers and supports a wide variety of other container implementations.

Agentless System Crawler (ASC) [5] is a monitoring tool with support for containers that collects monitoring information including performance metrics, system state, and configuration information. It provides the ability to build two types of plugins: function plugins for on-the-fly data aggregation or analysis, and output plugins for target monitoring and analytics endpoints.

There are many other tools which offer the ability to continuously collect metrics about running containers, however, if we were to continuously store and transmit these metrics, 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*.

2.5.3 Aggregation

Aggregation consists in the determination of important system wide properties in a decentralized manner, it is an essential building block towards monitoring distributed systems [7] [25]. It 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*.

Decomposability

For some aggregation functions, we may need to involve all elements in the multiset, however, for memory and bandwidth issues, it is impractical to perform a centralized computation, hence, the aim is to employ *in-transit computation*. In order to enable this, it is required that the aggregation function is **decomposable**.

Intuitively, a decomposable aggregation function is one where a function may be defined as a composition of other functions. Decomposable functions may **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 in the sum of all pairs divided by the count of peers that contributed to the aggregation.

Duplicate sensitiveness

The second property of aggregation is **duplicate sensitiveness**, and it is related to wether a given value occurs several times in a multiset. Depending on the aggregation function used, 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 [25].

	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

2.5.4 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.

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

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.

Continuous aggregation

Continuous aggregation consists in the continuous computation and exchange of partial averages data among all active nodes in the aggregation process [7]. This type of aggregation is attractive for gossip protocols, where nodes may employ varied gossip techniques to continuously share and update their values with random neighbors. Algorithms from this category are also attractive to use in edge environments, because they provide high accuracy while employing random unstructured overlays [23], consequently, the aggregation process retains the fault-tolerance and resilience to churn from these overlays.

2.5.5 Monitoring systems

We now discuss study popular monitoring systems in the literature, for each system we analyze its advantages and drawbacks, followed by a discussions with the systems' applicability to edge settings.

Astrolabe [43] is a distributed information management platform which 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.

Associated with each zone there is a Management Information Base (MIB), which consists in a set of attributes from 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 exception to the aforementioned mechanism, leaf zones contain *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, an *Information Request AFC*, 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, 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 which selects another agent at random and exchanges zone state with it. If the agents are within the same zone, they simply 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 the system administrator, which is a potential limitation towards scalability, given that configuration errors have the potential to heavily raise system latency and reduce traffic locality. Additionally, the original 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 single point of failure within each zone due to the use of representatives.

Ganglia [37] 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 which 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 situated 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) the project info page only claims scalability up to 2000 nodes.

SDIMS [57] (Scalable Distributed Information Management System) proposes a combination of techniques employed in Astrolabe [43] and distributed hash tables (in this case, Pastry [44]). It is based on an abstraction which exposes the aggregation trees provided by a DHT such as Pastry.

Given a key k, an aggregation tree is defined by the the union of the routing paths from all nodes to key k, where each routing step along the path to k corresponds to a level in the aggregation tree. Then, aggregation functions are associated an attribute type and 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, which enables load-balancing.

This achieves communication and memory efficiency when compared to gossip-based approaches, because MIBs have a lesser degree of replication, however, limitations which arise from employing SDIMS is that each node acts as an intermediate aggregation point for some attributes and as a leaf node for other attributes, which could potentially be a problem in edge settings, given that low-capacity nodes may become overloaded if they are intermediate aggregation points in multiple aggregation trees.

Prometheus [42] is an open-source monitoring and alerting toolkit originally built for recording any purely numeric time series. It supports machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures. This tool is especially useful for querying and collecting multi-dimensional data collections, it offers a platform towards configuring alerts, that trigger certain actions whenever a given criteria is met.

Prometheus allows federation, which consists in 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 which resembles 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.

2.5.6 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, and rely 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 [57] are unsuitable for edge environments, as devices are heavily constrained in memory and often are unreliable routers (which a DHT assumes all nodes can reliably do). Conversely, pure gossip systems such as Astrolabe [43] require heavy amounts of message exchanges to keep information up-to-date, and require manual configuration of the hierarchical tree, which may also be undesirable.

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 chapter 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 in the edge of the network.

3

GO-Babel

The first contribution of this masters dissertation is an event-based framework called GO-Babel. This framework is a port in Golang of Babel with a few additions focused on fault detection and latency probing. Babel itself is based on Yggdrasil , which in turn is inspired on .

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 needed features such as a fault detector and a latency measurement tool, we implemented a new version in Golang with these additions.

citation

citation

citation

citation

cite and discover paper of original event-based framework

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 for timers, request-reply patterns, notifications, and ease channel management.

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

In the figure 3.1 we may observe a high-level overview of the architecture of this framework, composed of five main components which communicate via callbacks. We now summarize each components' roles in the framework:

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.

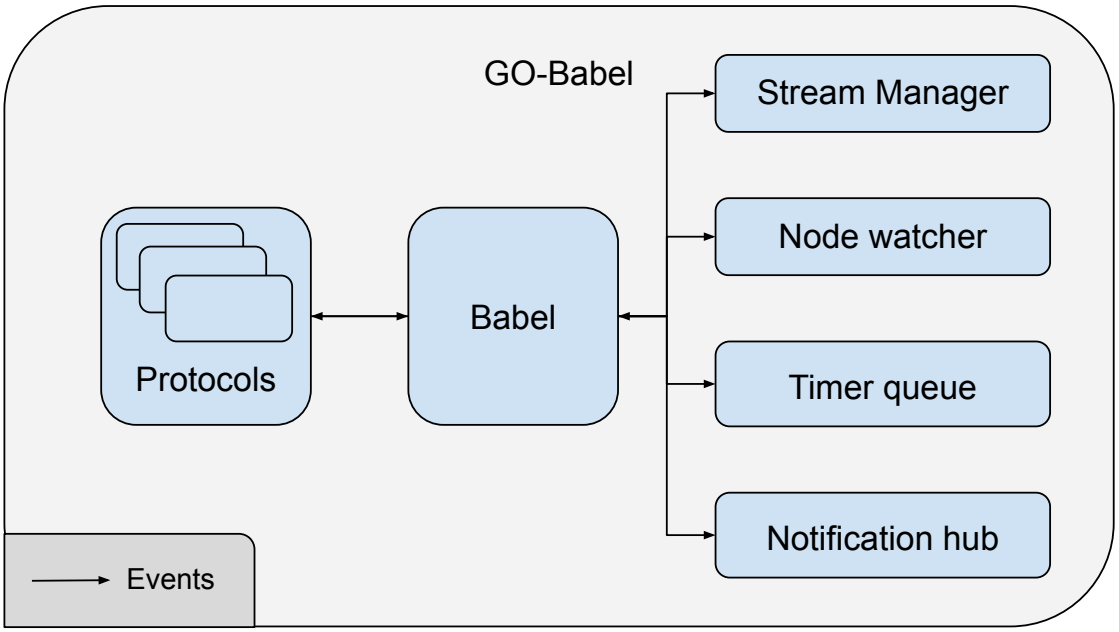

Figure 3.1: An overview of the architecture of GO-Babel

2. The stream manager is responsible for handling incoming and outgoing connections, connecting to new peers, and sending messages. Whenever the state of any connection changes, the stream manager delivers events to protocols with the connection status (e.g. if the connection established, connection failure, message sent/received, connection terminated, among others). It 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. The timer queue also allows creating periodic timers, which trigger at the set periodicity until cancelled.

4. The notification hub is responsible for handling notifications and notification subscriptions, propagating issued notifications to registered subscribers (protocols).

5. The node watcher allows for protocols to measure node latency and detect failures via a PHI-Accrual fault detector.

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 Babel and can found in .

insert citation

cite

3.2 Node Watcher

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

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, 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 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 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.

insert citation

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. If the watched node also does not accept the TCP connection, the node watcher sends a notification to the issuing protocol.

In order to prevent protocols from having to set timers to check the nodes' latency calculator or fault detector, the node watcher 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 Conclusion

We believe Go-Babel is a valuable contribution as it eases the implementation of self-improving protocols which employ latency as an optimization heuristic. In addition, it provides a secondary fault detector which may be employed together with the TCP connections. Lastly, as the implementation is in Golang , it allows easier integration with a range of packages already implemented in the language.

cite

Sinto que esta secçao nao devia existir?

4

DeMMON

DeMMon (Decentralized Management and Aggregation Overlay Network) is a monitoring framework which 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. **Have a partial set of nodes** from the system which are nearby (according to a certain proximity heuristic). These nodes are crucial in order to perform the aforementioned localized resource management decisions. In our framework, we chose latency as the heuristic for the proximity heuristic as 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. Ensure there are ways to **obtain the aggregate value of a metric distributed across the entire system** (e.g. the total number of nodes, service replicas, among others) without having to rely on a central component. This feature is crucial for resource management tools so they, for example, maintain a desired 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 decomissioning replicas, whenever the desired ratio of reaches a certain bound.

3. Having a way to perform **decentralized collection of metrics from "nearby"nodes**. This feature is useful for decentralized resource management systems as it allows nodes to perform actions such as load-balancing or QOS improvement: by collecting the metrics relative to the usage of nearby nodes, each node may decide (e.g

to perform latency, or reduce the load on a saturated service) to perform service migration or service replication.

4. As it is impossible to know ahead of time what information such systems would otherwise require, it is also a requirement to **be as flexible as possible in regard to the types of metrics** that are stored. This is paramount as resource management tools may need to store information in custom formats, tailored for their own needs.

5. Provide ways to efficiently **propagate information** accross nodes in the system. This is useful for resource management systems, as it allow them to disseminate information using the optimized connections established by the framework.

6. Ensure ways to **receive alerts** based on the collected information without resorting to periodically requesting/consulting it. By setting these alarms, resource management tools can, in turn, trigger resource management actions, for example, setting an alarm if the mean of the CPU usage over the last N seconds reaches a certain threshold, individual nodes may perform load-balancing actions.

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 fulfill the aforementioned requirements.

First, the **API** exposes the functionality of the framework, its main objectives are to: (1) allow issuing commands 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 (4 and 6) of the aforementioned requirements.

The **overlay network** strives to build 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

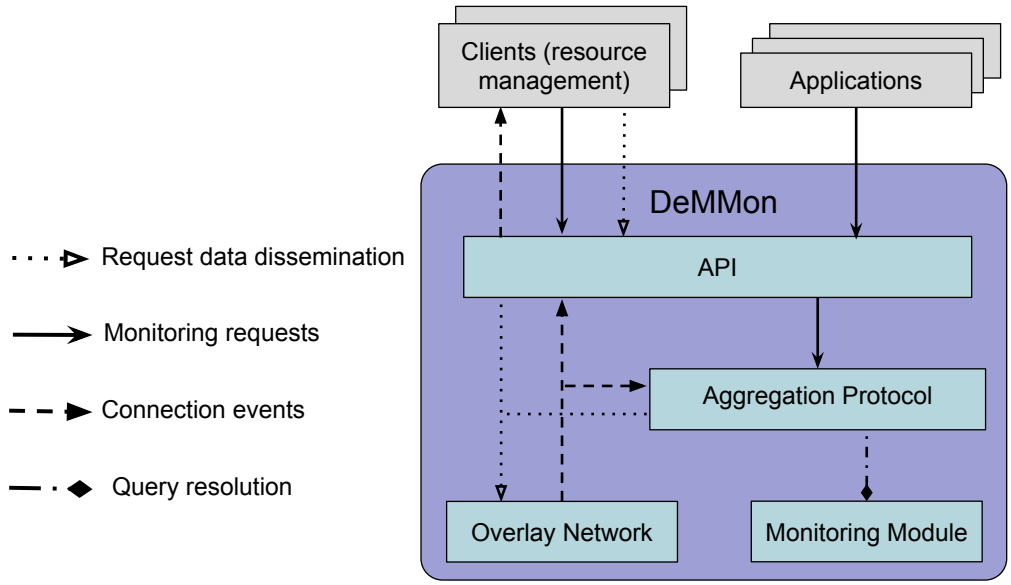

Figure 4.1: An overview of the architecture of DeMMon

of this protocol are the basis for the aggregation protocol. In addition, this module also offers limited horizont flood techniques, exposed through the API, fulfilling the points 1 and 5 of the requirements.

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 an aggregated global value (independently of being the root of the tree); and (3) neighborhood aggregation, where nodes collect values (non aggregated) of nearby nodes in term of hop proximity. These three mechanisms satisfy points 2 and 3 of the aforementioned requirements.

In the following sections we will provide a detailed explanation of each individual module, 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 overlay network, which aims to build and maintain a latency and capacity-aware tree-shaped network (capacity represents one, or a combination of, values that denote the node's computing and networking power). We begin by providing the considered system model, then follow with an overview of the mechanisms responsible for building and maintaining the tree. Lastly, we conclude the chapter with a summary and discussion of the protocol.

4.2.1 System Model

The assumed system model is assumed to be a distributed scenario composed of nodes connected to the internet set-up such that they can send and receive messages via the internet (with an external IP or port-forwarding). We also assume that nodes are spread throughout a large area and have varied capacity values.

Regarding the fault model, we assume that all but a small portion of nodes (also known as the landmarks, which in our model represent DCs) can fail, and when other nodes fail, they do so in a crash-fault manner, stopping all emissions and receptions of messages. We assume landmarks have additional fault tolerance given their privileged infrastructure, and additionally, we assume other such as replication [] mechanisms could be employed to ensure that faulty landmarks get replaced in case of failure.

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

4.2.2 Overview

As previously mentioned, the main objective of the created protocol is to establish a latency and capacity-aware multi-tree-shaped overlay network, rooted in the previously mentioned landmarks. Our motivations for choosing the tree structure for the network are the following: (1) to map the cloud-edge environment, by rooting the trees on nodes running DCs in the cloud, and creating a hierarchical structure for other, less powerful, nodes to 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 ; 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.

isto e esticar?

add ref

In order to ease the explanation of the protocol, it is important to define some terms which we will use frequently to explain the devised protocol. The tree structure the protocol aims to establish and maintain is observable in figure , which, as previously referenced, is composed of multiple interconnected trees. The nodes connected to the landmarks (denoted their **children**) may themselves be the parent of their own children, which would have the landmark as their **grandparent**. Intuitively, the **descendants** of a node are all of its children and children's children, recursively, until the leaves. 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 connection may be called its **active view**.

criar imagem para ilustrar estrutura resultante

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 bounding 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 choosing the initial parent connection, which performs a greedy depth-first search to find a suitable low latency node in the network with more than zero children. This mechanism is the first to be executed by all nodes in the system, with the pseudocode presented in algorithm 1.

Its first step (line 4) is to initialize the state of the joining node, composed by: (1) a map of type Node containing all successfully contacted nodes so far the join process, (2) a collection of type Node and a set of timer ids for each contacted node, (3) the best node contacted so far in the join process, (4) a timer id for contacting the chosen node in the join process, and finally (5) a variable of type Node denoting the peer itself. The type "Node" is a collection of attributes regarding a node, composed of: (1) latency measured, (2) its current parent, (3) number of children, (4) whether the node replied to the message, (5) its IP, (6) an array of coordinates (denoting its measured latency to each landmark, used in passive view maintenance mechanism), and finally, (7) an array of its childrens' IP and their number of children.

Then, each node joins the system, the procedures taken to join the tree differ consonant the node is a landmark or not. Given that landmarks are the roots of the trees, they have no parent in the resulting overlay, and consequently, in the join algorithm, these nodes attempt to repeatedly establish a connection with other landmarks by sending a special message. Landmarks that receive this message will send a reply and establish a connection back (line 16). Any joining landmark node only stops sending messages to other landmarks when the respective reply is received.

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                                          ▷ nodes being contacted
7:      landmarks                                               ▷ landmark nodes
8:      joinTimeouts                                            ▷ collection of contacted nodes -> timerIDs
9:      bestPeerLastLevel : Node                                ▷ the best peer contacted so far in the join process
10:     joinReqTimeoutTid                                       ▷ timerID for join messages
11:     self : Node                                             ▷ myself
12:
13: Upon Init(landmarks : IP[ Do, selfIP, isLandmark)]
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>, p.IP)
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,IIP,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:
```

Nodes that are not landmarks begin the process of choosing their initial parent, initiated by sending a JOIN message via a temporary TCP channel, measuring the latency, and issuing "joinTimers" for all tree roots (line 17), then the node awaits the responses from the contacted nodes, during this process, the joining node listens for any "joinTimers" which have triggered, or until any of the node measurements has been unsuccessful (meaning contacted nodes have exceeded their reply timeout), if this happens, in the case of the contacted node being a landmark, the joining node aborts the join process and waits a configurable amount of time until attempting to re-join the overlay again. If the timed-out node is not a landmark, then that node is excluded from the remaining of the join process, and the join process is resumed as normal (line 51).

When a node receives a JOIN message, it sends a JOINREPLY message back to the original sender containing: its parent, itself, and its children (line 19). When the joining node receives the joinReply, it discards those that are from a timed-out node or from any node whose parent was not contacted in the join process (the node changed parent during the join process). Then, whenever the joining node has either: received the JOINREPLY messages from all contacted nodes and stored the information (line 22), or they have been timed-out via the "joinTimers", it evaluates all contacted nodes, attempting to find the contacted node with the lowest latency which is a suitable parent, by performing the following verifications:

1. Verify if the node already has any children or if the node is a landmark (and can become the parent of the joining node) (line 38).

2. Verify if there was a node already contacted previously which was a suitable parent and had lower latency, in case there was, the joining node sends a "JoinRequest" and sets up a "JoinRequestTimer" for that node, and stops the verification process. (line 40)

3. Verify if the current node has both enough children, and has the lowest latency up to this point in the join process, 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 children of that node which themselves have more than one children (line 43). Note that if none the current nodes' children are suitable parents (i.e. have no children themselves), then the condition in line 35 is triggered and the joining node will request the current best node to be its parent.

If none of the verified peers was suitable to start a new recursive step (line 48) (either had no children or all verified nodes had higher latency than a previously contacted node), then the node joining node sends a "JoinRequest" to that node and sets up a "JoinRequestTimer" for the best previously contacted node (any node which receives a "JoinRequest" message replies with a "JoinRequestReply").

The join process is concluded with both the reception of a "JoinRequestReply" and the establishment of the connection between the sender and receiver of the message.

If the "JoinRequestTimer" timer triggers while waiting for the response, the node will recursively fall back to the parent of the selected node or re-join the overlay later in case there is no parent available.

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. In sum, this mechanism is coordinated by each parent and achieved via sending messages to some of its children signalling that they should connect to another specified parent. It achieves this by choosing new parents to form new groups using latency and node capacity as heuristics for the parent choice, where the information necessary to employ these two heuristics is obtained via periodic transmission from every child to its parent. This mechanism only executes when a group exceeds its size limit and attempts to keep group sized near the maximum configured limit.

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

The mechanism starts with the propagation of information from the parent to the children and vice-versa. As observable in lines 7-16), each parent transmits to its children a list of its current siblings, and propagates to its parent the latency to each of its siblings. Then, when this information is received (lines 17 and 24), it is merged into their local states for later use.

The second part of this mechanism is also periodic and is responsible for maintaining the group sizes by creating new parents or by sending children to already created groups (line 29). This mechanism is only executed if the number of children of a certain node (denoted the "proposer") exceeds the configured maximum number of children per parent. In this mechanism, a proposer node proposes to one of its children (denoted node "A") a change of parent to another one of its children, (denoted the "proposed" 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 33-38). Then, the node iterates the added edges set by ascending order of latency, performing the following verifications:

1. If the number of current children minus the nodes already sent to a lower level is lower than the maximum size of a group, then the node concludes the mechanism (line 44)

2. If any of the two nodes were already sent to lower levels of the tree, then the current edge is skipped (line 46).

Algorithm 2 Membership protocol (Active view Optimization)

```
1:  State
2:      parent                                                              ▷ defined in join
3:      children                                                            ▷ defined in join
4:      siblings
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, siblingLatencies>, 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:
24: Upon receive(UpdateChildStatus<child, childSiblingLatencies>, sender) Do
25:     if children[sender] != nil then
26:         children[sender] ⟵ child
27:         childrenLatencies[sender] ⟵ childSiblingLatencies
28:
29: Every config.evalGroupSize Do
30:     if len(children) <= config.maxGroupSize then
31:         return
32:     childrenLatValues ⟵ set()
33:     for c1 in children do
34:         for <c2, lat> in childrenLatencies[c] do
35:             if lat - c1.measuredLatency > d.config.maxLatDowngrade then
36:                 continue
37:             if c1.cap > c2.cap then childrenLatValues.add(<c1,c2,lat>)
38:             else childrenLatValues.add(<c2,c1,lat>)
39:     kickedNodes, newParents ⟵ set(),set()
40:     pChildren ⟵ dict<string,set<Node>>                                  ▷ set of potential children for each children
41:     sortByLatency(childrenLatValues)
42:     idealGroupSize ⟵ config.maxSize - config.MinGroupSize
43:     for <c1,c2,lat> in childrenLatValues do
44:         if len(children) - len(kickedNodes) <= config.maxSize then
45:             break
46:         if c1 ∈ kickedNodes || c2 ∈ kickedNodes || c1 ∈ newParents then
47:             continue
48:         if c1.nrChildren == 0 && newParents[c1] == nil then             ▷ Node is not yet a parent
49:             pChildren[c1] ⟵ pChildren[c1] + c2
50:             if len(pChildren) == config.MinGroupSize then
51:                 for potentialChild in pChildren[c1] do
52:                     newParents ⟵ newParents + c1
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:         else
59:             kickedNodes ⟵ kickedNodes + c2
60:             send(OptimizationPropose<higherCapNode>, lowerCapNode)
61:
62: Upon receive(OptimizationPropose<newParent>, sender) Do
63:     if sender == parent then
64:         send(OptimizationProposeRequest<sender>, newParent)
65:
66: Upon receive(OptimizationProposeRequest<p>, sender) Do
67:     if  p == parent && sender in siblings then                          ▷ parent issuing the message is my parent
68:         addChild(sender)
69:         send(OptimizationProposeRequestReply<true,p>, sender)
70:     else
71:         sendSideChannel(OptimizationProposeRequestReply<false,p>, sender)
72:
73: Upon receive(OptimizationProposeRequestReply<reply,p>, sender) Do
74:     if parent == p then
75:         if reply then
76:             sendMessageAndDisconnectFrom(DisconnectMessage<>, parent)
77:             addParent(sender)
78:     else
79:         sendMessageTemporaryConn(DisconnectMessage<>, p)
80:
```

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 49). When this set has the same size as the minimum configured group size, then the node issues "OptimizationPropose" messages for each node of the set, and removes each child from every other node's potential children (lines 51-57). 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 a "OptimizationPropose" message to it (line58).

When node "A" receives an "OptimizationPropose" message with a new proposed parent (line 62), 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 node to the proposed parent, signalling it wishes to become its child. Then, when the proposed parent receives the message (line 66), it verifies that the proposer node is still its parent and that node "A" is also its sibling, if yes, then it adds the node as its new child and replies with an "OptimizationProposeRequestReply", which contains a boolean flag, signalling if the node was added as a child or not. Lastly, when this message is received (line 73), the node 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 proposed node becomes the parent of node "A", 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), and when possible, node "A" obtains a new node with lower latency than its current latency to the proposer node.

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

ref to code

It is important to mention a final mechanism in active maintenance which is ommited from the pseudocode. This mechanism is responsible for ensuring that groups sizes do not become too small, according to a configuration parameter. In sum, every node periodically verifies the number of peers which are its siblings, if this number is lower than a certain threshold, the node "rolls a dice" (essentially generates a random number and verifies if it is lower than a certain threshold) to decide if it should abandon the current group. If the generated number is lower than the threshold, the node sends a message to its grandparent asking to become its child. When the grandparent receives the message, it adds the node to its children and sends a message reply, signalling to the original node that it was accepted as a child. 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) decreases quadratically to the difference of the configured group size and the current node's group size.

4.2.2.3 Passive view maintenance & Oportunistic improvement

The third mechanism of the devised membership algorithm is the passive view maintenance mechanism, it is responsible for creating an auxiliary pool of nodes in the overlay which are not descendants of the executing node. When full, the 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.

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 following mechanism. We now explain each in detail, and how it is obtained:

1. The ID of each node is a collection of ID 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 a possible ID would be: AAA/BBB/CCC, where the ID segments are: "AAA" ,"BBB" and "CCC", this gives each node enough information based on an ID to evaluate if any other node in the overlay is its a descendent, therefore allowing nodes to evaluate if a change of parent in the overlay causes a cycle in the tree. This ID structure also allows nodes check what is the level of any node (the number of segments of the ID is the same as the level of a node in the tree).

2. The coordinates are an array of integers representing the latency every node measured to all landmarks, these coordinates are used as a heuristic for measuring new nodes in the passive view which are potential parents.

3. The version of a node is a monotonic integer which is incremented at every ID change and child addition or removal. This version is used in random walks, to update peers which are currently in the passive view with their new IDs, which prevents nodes from attempting to measure nodes which would be incompatible parents (i.e. they are their descendants, or have no children themselves)

With these concepts explained, we now present the pseudocode (algorithm 4.2.2.3) for the mechanism. The first lines declare the new necessary state to the mechanism, which is composed of a set of nodes denoting the passive view of the node (line 2). Then, in the following lines we may observe the mechanism for filling the passive view, this is a periodic procedure which triggers the emmision of new random walk messages, triggered at pre-configured intervals (line 4), the created random walk message contains a random sample of nodes from the passive view and the active view, the original sender's ID, and

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 9), if the message has travelled a certain number of configurable hops, then the receiving node removes a configurable number of nodes from the sample, if the message has not yet travelled the number of hops, the previous step is skipped. Then, the node merges the removed nodes into his passive view, and adds a random sample of nodes from his own passive and active view to the sample (discarding nodes previously in the sample if the configured maximum sample size is exceeded) (lines 13-22). The intuition behind skipping a certain number of hops before removing nodes from the sample is to promote exchanges of information with nodes further away (in terms of hops) from the original sender. 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 which is not a descendant of the original sender, if there is no such node, or the TTL is zero, then the node sends, via a temporary connection, a "RandomWalkReply" message to the original sender of the random walk with the sample (lines 24-27). 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 29-33).

As the overlay evolves with time, the passive views of nodes fill with nodes that are not descendants of the node in question, given this, they are suitable for latency optimizations and fault recovery (in case a parent dies). The procedure responsible for evaluating the nodes for latency optimizations (lines 35) is also evaluated periodically at pre-configured intervals, in this procedure, the node selects a random sample of nodes and another sample based on the euclidean distance of their coordinates to the measuring nodes' (with configurable maximum size). Each node selected for this sample (candidates) must satisfy the following conditions (lines 40 and 45):

1. If the candidate has no children, then it is excluded from the process.

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 (in order to prevent nodes with multiple children from going down in levels and favour instead nodes with no children joining the upper levels of the tree).

After the measurements are issued, whenever a "peerMeasured" event is triggered, the node compares the current latency of its parent with the measured nodes' latency: if the latency to the measured node is lower than the current parents' by a configurable threshold, then the measuring node will send an "OportunisticImprovementReq" message to the measured node. When it receives this message, 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.

Algorithm 3 Membership protocol (Passive view maintenance)

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

4.2.2.4 Fault tolerance

Fault tolerance in the protocol is done whenever a parent failure is detected, either due to the PHI-accrual failure detector provided by the Node Watcher (3.2) or by failure of a TCP connection which triggers a notification to the protocol. The node first attempts to fall back to its grandparent (provided via the periodic information in 4.2.2.2), then, if this fails, it falls back to any node in its passive view that is not a descendent. Fault recovery is achieved by sending a "FaultRecovery" message containing its ID and setting 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 removed from the passive view. If the passive view becomes empty, then the node starts the join mechanism again (subsection 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 specified the information which is exchanged with it over time, and the parent eploys this information to coordinate with its children in an attempt to maintain the group size within a certain bound, and attempt reduce overall system latency in the process. Lastly, we explained how nodes obtain information about other random nodes in the nertwork, and how that information is used to perform latency optimizations which reduce the total overlay network latency.

4.3 Aggregation protocol

Provided with a membership protocol capable of coordinating nodes into building an efficient tree structure, in this section, we now discuss how we leveraged on 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. The three primitives provided are the following: (1) tree aggregation, (2) neighbourhood aggregation, and finally, (3) global aggregation, which we now clarify in further detail, starting with tree aggregation.

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) it forms one or more tree-shaped networks, whose roots are interconnected. Nodes in these trees must be connected to their parents, children and siblings (also denoted by their active view) in a bidirectional manner. The overlay protocol must also provide events for each node that is added or removed from the nodes' active view.

4.3.1 Tree aggregation

Tree aggregation is the mechanism responsible for collecting metrics and merging them using the tree, collecting 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 is executable by all nodes in the mechanism, and if two different nodes are aggregating the same values with the same parameters, and a node is descendent of the other, then the descendant node will (when possible) reuse the values of the already existing tree by embedding its tree into the ascendants'.

The pseudocode for this mechanism (defined in 4) begins by defining the necessary state to execute the 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" containing the necessary metadata for each aggregation tree, each value of this map is composed by: (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, (5) a boolean value representing if the value should be exported locally, (6) a boolean representing if the parent is also in the tree (and the node must propagate values to it or not), and finally (7) the ID of the tree from the parent's perspective (or nil, if the parent is not in the tree). The second map denominated "lastSeen", contains a Timestamp for each tree, representing the last time the parent sent a message for that tree, and finally, the "childValues" map which contains, for each tree, the values emitted by the children and the timestamp of their reception.

Nodes begin executing this mechanism when the API sends a request to the protocol (line 7), which contains the maximum height of the tree, the merge function, the query to obtain the local value, the periodicity to execute the mechanism, and the resulting metric name. Upon the reception of this request, the node creates the ID for the aggregation tree by hashing the concatenation of the tree height, the merge function, the query, the mechanism periodicity and the resulting metric name, guaranteeing 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 tree's aggregation results, not increasing the necessary messages to collect the metric values for an additional node. After this, the node receiving the request adds the tree to its local aggregation "tIds" map, first checking if it is already federated to a 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 locally to true. Otherwise, it adds a new entry to the "tIds" map and sets up a periodic timer for that aggregation tree (lines 8 to 14).

In order to federate all nodes into their trees, nodes periodically (using configured intervals) broadcast to their children a "Subscription" message containing the aggregation

Algorithm 4 Tree aggregation

```
1:  State
2:      parent,children,siblings                                  ▷ Defined by the overlay protocol
3:      tIds ⟵ map()
4:      lastSeen ⟵ map()
5:      childValues ⟵ map()
6:
7:  Upon StartTreeAggregationRequest(tHeight, mergeF, query, periodicity ,outmName) Do
8:      tId ⟵ hash(tHeight + mergeF + query + periodicity + outmName)
9:      if tId in tIds then
10:         <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
11:         tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, true, isParentSub, ptId>
12:     else:
13:         timerId ⟵ registerPeriodicTimer(ExportTreeAggTimer(tId), periodicity)
14:         tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, true, false, nil>
15:
16: Upon ExportTreeAggTimer(tId) Do
17:     <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, isParentSub, ptId> ⟵ tIds[tId]
18:     if isParentSub && timeSince(tIdLastSeen[tId]) > config.treeAggExpiration then
19:         if !isLocal then
20:             tIds.delete(tId)
21:             lastSeen.delete(tId)
22:             cancelTimer(timerId)
23:             return
24:         else
25:             tIds[tId] ⟵ <tHeight, mergeF, query, periodicity, outmName, timerId, isLocal, false, nil>
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:
```

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 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. 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 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 not send more "Subscription" messages.

Whenever the aforementioned periodic timer called "ExportTreeAggTimer" triggers, (line 16), the node checks if the respective aggregation tree has expired (i.e. if the parent stopped refreshing the aggregation tree), if it has, and the node is not a root of the tree, then it cancels the timer and deletes any related tree metadata, if the node is a root of the tree, it sets the flag representing wether the node should propagate to the parent as "false". If the tree has not expired, the node evaluates the 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 configured to do so, it will store the value locally, and if the flag signalling it should propagate to the parent has the value "true", it sends a message to the parent containing the obtained value. Upon the reception of this 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 finally, stores the received value into the "childValues" map.

Although it is omitted from the pseudocode, it is important 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). This feature is useful for applications as it provides them with a sense of directionality. For example, if an application needed to deploy service replicas according to geographical proximity to a certain target, this feature can be useful for obtaining an average of latitudes and longitudes for all nodes "behind" a certain node in the active view. Then, an application may recursively send messages to nodes in the immediate view that approximate a certain geographical target.

maybe imagem?

talvez meter uma conclusão aqui?

4.3.2 Neighborhood 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 certain node needs to perform service replication/migration, it may collect metrics related to the capacity and usage 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 Pub-Sub system, where nodes create an aggregation trees rooted upon themselves by broadcasting "Subscription" messages periodically with a configurable hop-based range (or TTL). Nodes that receive this message become federated in the tree, and rebroadcast it (up to the configured TTL) to other peers, federating all nodes in the configured hop-range. Afterwards, for each tree, all federated nodes periodically propagate their locally obtained values towards the root of each tree using the reverse path established by the broadcast message.

Trees have associated IDs, generated using hashing in a similar manner to the tree algorithm defined in 4.3.1 (without using the level in the hash process), and consequently, all nodes collecting the same metric using this mechanism will have trees with equal IDs. Nodes belonging to overlapping trees (i.e. in the 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, in order to prevent unnecessary query evaluations. In addition, nodes in overlapping trees, when possible, also deduplicate "Subscription" messages.

This mechanism, similarly to Tree Aggregation (subsection 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 4.2) 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 periodically a "Subscription" message to its immediate neighbours (illustrated in step 2 of figure 4.2) containing the ID of the tree, the TTL, the query to obtain their local values, and the mechanism periodicity.

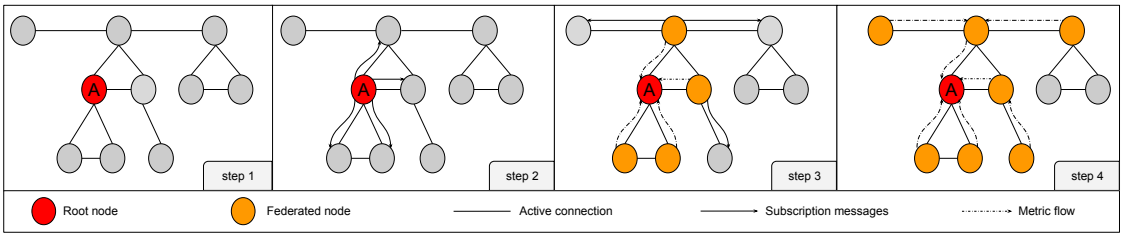

Figure 4.2: Neighborhood aggregation subscription process (TTL=2)

Whenever a node receives the "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, then the timestamp is refreshed.
3. Decreases the TTL of the message by one
4. If the message TTL is 0, the node returns from the 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 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 received to these nodes in the last (configurable) duration. If it has, the sender skips the broadcast for that node.

With this, in case two different nodes in the system are collecting the same metrics on the same periodicity, the "Subscription" messages are not sent unnecessarily to nodes already subscribed to that tree, because whenever the second "Subscription" message arrives, it is not broadcasted unless a certain amount of time has passed. This process is illustrated by node A in figure 4.3, where node A receives the Subscribe 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.

After nodes become federated in 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. This process is illustrated on pictures 4.3.2 4.3.2 and 4.3.2). Nodes that receive this message increment their hop counter and forward it to each other node in its active view that has sent a "Subscription" message with a TTL higher or equal to the hop count within a (configurable) time frame.

Lastly, nodes periodically verify, for each tree, the time passed since the reception of the last "Subscribe" message and remove the ones received after more than a (configurable) amount of time. When nodes remove the expired entry, if there is no other entry for that tree ID, they also stop propagating metric values related to that tree (illustrated in fig. 4.4).

completar com summary

4.3.3 Global aggregation

Global aggregation is the mechanism executed whenever a certain node wishes (via requests from the API) to obtain a summarized global view of the system (e.g. the total

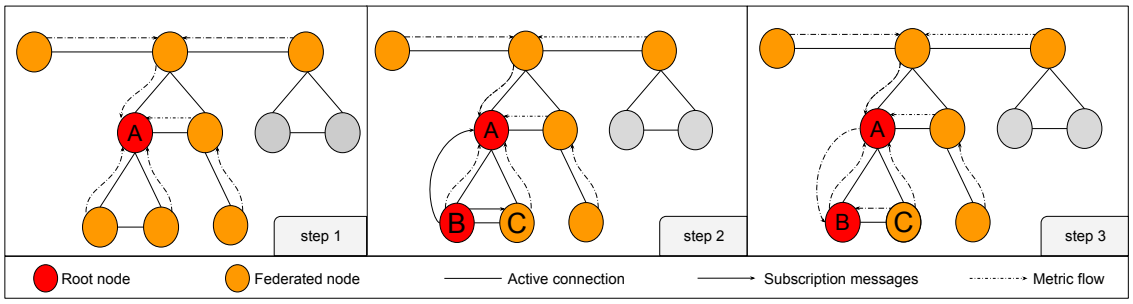

Figure 4.3: Neighborhood aggregation second subscribe (TTL=2)

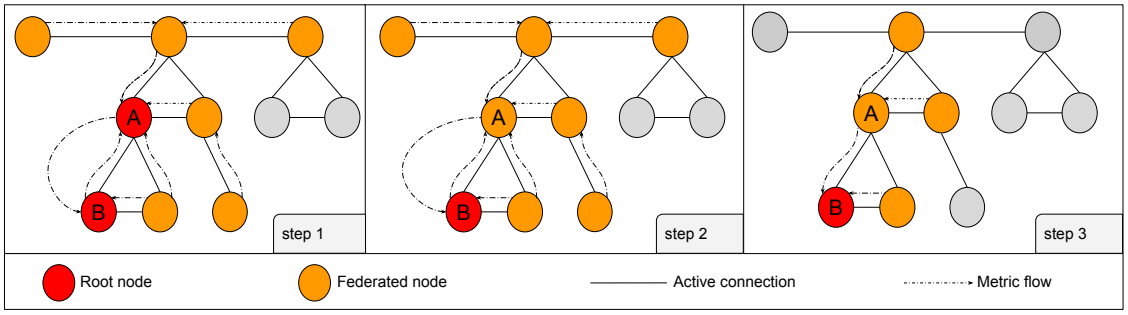

Figure 4.4: Neighborhood unsubscribe (TTL=2)

number of nodes in the system). This process, similarly to 4.3.1 and 4.3.2, is based on federating nodes of the system into **aggregation trees**, nodes federated in these trees who are not the roots are denoted **aggregator** nodes, as their role is to collect their neighbours' metric values, aggregate them, and send them toward the root of the tree.

In global aggregation, all nodes participate in all trees, either as a root (if they wish to collect the globally aggregated value), or alternatively as aggregator nodes. In this mechanism, if a tree has multiple root nodes, then the metric values are reused (when possible) from the first tree root towards the other roots and tree maintenance mechanisms of the aggregator trees are deduplicated.

citar akos

This mechanism is inspired in the work from , 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 via every node periodically broadcasting to every neighbouring peer their locally aggregated value minus each neighour's contribution, and merging all received contributions with the locally generated one, the continuous exeution 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 work, instead of only performing aggregation of a single metric value, we generalize the approach to allow the on-demand creation and teardown of multiple trees, potentially rooted in multiple nodes, with all roots collecting the globally aggregated value. The relaxation of these constraints creates additional challenges regarding the transmission of redundant messages (i.e. to maintain the aggregation tree), which we attempt to mitigate in this work. It is important to mention that, in a scenario where all nodes of the system are roots of the aggregation system, this mechanism behaves similarly to , with the only difference being of allowing the on-demand start and decommission of the aggregation process.

citar akos

The state necessary for the execution of this mechanism (defined in alg. 5 lines 1 to 5) 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 emmitted by the overlay protocol (omitted from the pseudocode). In addition, the state contains a map denominated "tIds", this map holds the metadata needed to manage the aggregation trees,composed of: (1) a difference function, which is 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, the query to obtain the local values, (3) the query to obtain the local values, (4) the resulting output name for the aggregated metric, (5) the periodicity 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 receicing values for that tree (previously mentioned as the aggregator nodes). Lastly, the state also contains a map denominated "lastTimeSent", whose values contain for each tree and neighboring node, the last time the executing node has sent a message refreshing the corresponding tree existence to it and a map called "neighValues", which stores the propagated neighbour values and a timestamp of the reception of these values.

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

4.3.3.1 Tree creation and maintenance

As previously mentioned, global aggregation allows the on-demand creation and decommission of aggregation trees. This process is achieved through a time-based mechanism, where nodes periodically refresh the trees they are the roots of via message broadcasts which are then forwarded by other nodes in the system until it reaches every node. If an aggregator node (not a root) does not receive a message from any neighbour refreshing the existence of a tree within a certain time frame, it decomissions the tree locally.

This process is defined in alg. 5 lines 19 to 30), 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 clears all entries in the "aggNeighs" which 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, the tree is removed.

Whenever the "RefreshGaggTree" message is received (alg. 5 line 32), the receiver adds the previously unknown trees into its local "tIds" map and sets up a periodic "ExportGlobalAggTimer" for each added tree (lines 35 to 40). 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 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 in order to prevent trees from being refreshed multiple times unnecessarily (as the receiving node will propagate a message refreshing the same trees later, or has refreshed it in the last seconds).

4.3.3.2 Metric propagation

With the tree maintenance 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 51), 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 vy the "isLocal" flag (lines 52 to 57). 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 58 to 59).

Finally, whenever nodes receive the "PropagateGAggValues" (alg. 5 line 51) message 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 one of the conditions does not verify, and store the propagated value locally in the "neighValues" map, for later use in computing the aggregated value.

In sum, nodes who 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 the"RefreshGaggTree" 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 , multiple nodes in the system can simultaneously obtain the aggregated value efficiently and in a decentralized manner.

cite akos again?

4.3.4 Summary

In this section we presented the devised aggregation protocol which leverages on 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 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. Nodes federated in these trees periodically merge their local value with their childrens' and send a message containing it to their parent.

The second primitive is **neighborhood 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 on themselves. Nodes in this tree 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 (in case multiple neighboring nodes have sent federation messages).

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 which corresponds to the globally aggregated value of the system. This primitive is inspired by work in 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 on-demand (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 decomission 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, **neighborhood 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.

4.4 Monitoring module

As previously mentioned, 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. It is 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 abstractions for a proof of concept, and aspects such as the storage of metrics in disk or the efficiency of the query language are engineering challenges that are orthogonal to the conducted work.

This module is composed by three different components (illustrated by fig. 4.5): 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**, which allows the insertion and retrieval of time-series data. This time-series database makes use of an in-memory index to efficiently retrieve and insert data into the corresponding time-series.

2. The **query engine** is the component tasked with resolving queries to the time-series database, in sum, it holds a set of sandboxes that evaluate user-provisioned queries that extract and apply aggregation functions to metrics present in the time-series database.

3. Finally, the **alert manager** manages the alarms issued by the API, these alarms contain a condition that, when issued and triggered, should emit a notification to the issuing client. In sum, this component periodically verifies the issued queries using the **query engine** and propagates an event to the client whenever the condition is verified.

In order to ease the explanation of these components, it is important to first describe what is the structure of the time-series data used in this framerwork, which has a similar structure to the metric types of InfluxDB . cite

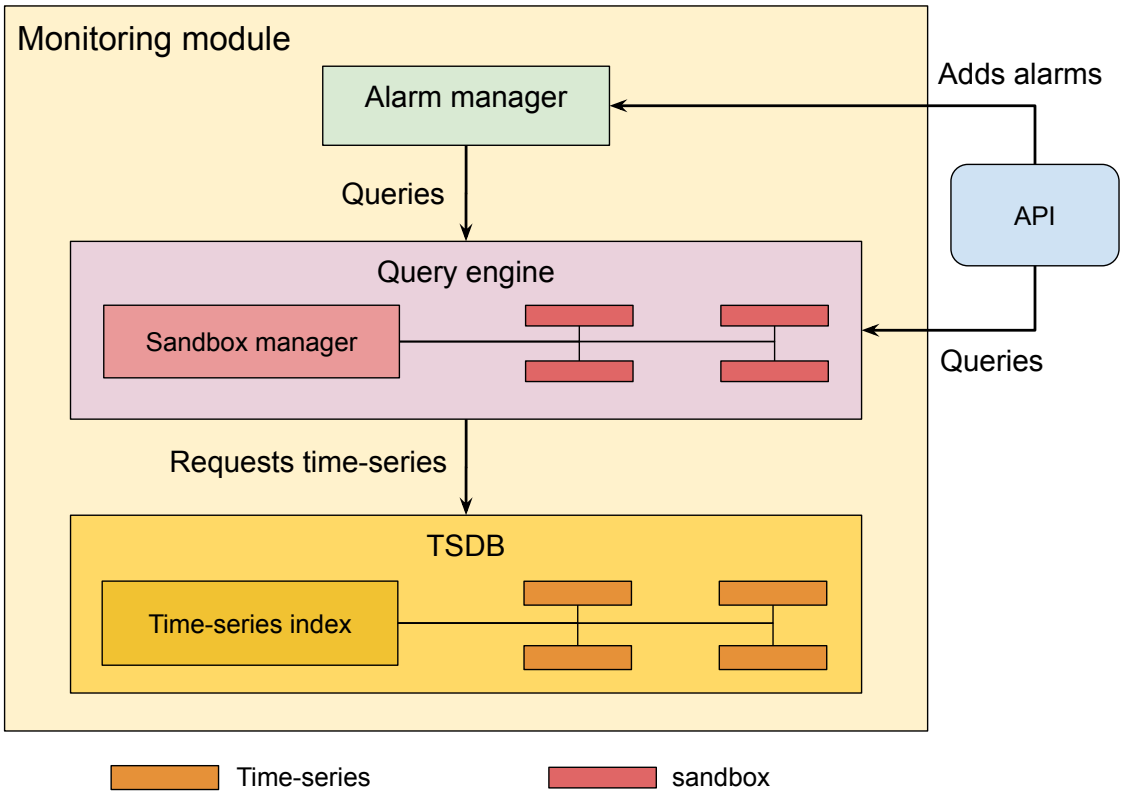

Figure 4.5: An overview of the monitoring module

4.4.0.1 Metric structure

In DeMMon, a metric is composed by four elements: first, the **name**, which is a string containing the name of information that is being stored, it should be a human-readable name which is self-describing (e.g. "CPU-USAGE"), second, the metric **tags**, that are a set of string pairs which denote attributes related to the metric that is stored, such as 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**, which 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".

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 marshallable using the JSON package provided by the golang package) . This allows the devised framework to represent a multitude of different information types, such as histograms, strings, string maps, among others. For example, a decentralized service management system aimed at deploying service replicas in close proximity to the clients, may use, for example, a histogram of geographical locations, with pre-determined classes. This way, this system would have a data structure which would ease the process

cite

of finding a node in a certain 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, which in this system, are stored only in-memory. In this system, these sequences are indexed as a function of their name, periodicity, 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, this is essentially a component that holds all time-series data with a certain name, periodicity and capacity. The periodicity of a bucket 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. This allows the system to pre-allocate the memory necessary (using an array) for each time-series upon its creation.

Within a bucket, metrics are stored in a map and indexed by their tags, this is done by creating a key which creates the same key for each similar tag set, independent of its order: whenever a metric is inserted, its tags pairs are sorted alphabetically (by their key), and are 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 not one prevously in the system).

Time-series advance time in an on-demand manner, which means that any time-series, before returning values for a read or write request, 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 time series iterates its points from the oldest to the newest point, and removes all points which have exceeded its time window. It is important to mention that series concurrency is maintained using locking mechanisms, where operations which do not affect the state of the time series are executed concurrently, and operations which would otherwise change the time-series state 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 querying modules 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 interpreters to perform this work. This means that queries are essentially javascript code, meaning that users have infinite control on the behavior of their queries, provided these dont exceed the query timeout.

In order to provide this functionality we opted for using the package Otto , this insert citation

package provides access to javascript "virtual machines" that essentially take a string containing a javascript script, and build an AST from the parsed code, then this AST is executed and the result is returned by the VM, which allows a practically infinite range of query options. In order to allow users to access the time-series stored in the TSDB without directly doing so (as the user-defined queries could potentially alter internal state of time time-series), the query engine provides every Otto virtual machine access to the following functions, which return time-series from the database:

1. SelectLast(Bucket_Name, <Tag_set_regex>), this function returns the last point for every time-series which are in the supplied bucket that match 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 woud be, for example: "selectLast(CPU_USAGE, <host:.*, cluster:cluster1>)"

2. Select(Bucket_Name, <Tag_set_regex) this function behaves similarly to selectLast, however it returns all points in all matched time series.

3. SelectRange(Bucket_Name, <Tag_set_regex>, startDate, endDate), this function behaves similarly to select and selectLast, however its arguments take an additional time range, and instead of only returning the last value it returns all points inside the supplied range.

With these 3 functions, clients can select either a partial set of points or the totality of points from every time series stored in the system, these time series are then usable in the javascript code.

We believe covers 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 timeseries 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" (this can be omitted if the query simply returns a set of values). Any query executed in deMMon can only result in one of two options: a single time series or an array of time series. Given this, in order to allow the creation of new time series during the query process, there are two additional functions available to the virtual machines: the first is called "NewTimeSeries", which creates a new time series, this function takes as argumments the name, tags and values which will integrate the time series; second, we have the function called "NewObservable" which takes the observed value and a timestamp and to create 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.

4.

meter mais um exemplo de uma query custom

With this, clients are able to, through the API, 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 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 which contain, among others, a condition to observe (e.g. the percentage of CPU usage). This component is essentially responsible for periodically verifying this condition and issuing notifications to the client whenever the condition is observed. Alarms are paramount to prevent applications from having to periodically access the API to verify the condition themselves, effectively saving bandwidth.

In deMMon, an alarm is composed by the following components:

1. **Query** - This denotes the query to perform periodically, this query must return a boolean value.

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

3. **Backoff time** - The backoff time is a duration decreases 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 alarm periodicity is low).

4. The **watch list** is a set which, for every item, contains both a name and a set of tag filters. Whenever the alarm manager receivers an alarm containing a watchList, 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. The **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, effectively saving CPU time. This option is meant to be used together with the watch list, for example, for checking a parameter which is rarely altered. It is important to mention that if this parameter is set to false, then time-based effects such as the expiration of either time series or points are not observed (as the alarm is not checked periodically).

The monitoring module, whenever it receives a new alarm, essentially adds it to a priority queue containing all the alarms which uses the time of reception of the alarms plus their periodicity as their key to the queue. Using this data structure, alarms are sorted by the time at which they need to be verified. Then, the monitoring module continuosly obtains and removes the first item of the queue, containing the next alarm to verify out of all issued alarms. After this, monitoring module waits until it is time of verification of that alarm (i.e. the time of reception of the alarm plus its periodicity), then verifies it, and adds it to the queue with a new timestamp corresponding to the current time plus the alarms' periodicity.

Whenever the alarm is verified and the result of its query returns "true", the alarm manager verifies if it has emitted a notification for that alarm in the last "Backoff time", and issues a notification to the client if it hasn't.

4.5 API

The API is the last module of the devised framework. As previously mentioned, 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 detail its most relevant operations.

4.5.1 Overview

citation?

This API is coalesced by a message-based protocol performed via WebSockets . 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. 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 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 server, which contains a single

endpoint, 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 . put repo in

When connected, clients and servers exchange JSON formatted messages which may contain messages related to the behaviour of 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 sporadic messages to the client containing events related to the issued subscription. This type of operation is used, 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 what we believe to be 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 section , buckets are containers for all time-series data with a certain name, periodicity ref 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. The insertion of values is performed using messages the metric values. When these messages 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 devised 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 view of the deMMon server, it sends a message containing the change type (if either a new connection was established to a node, or if a previous connection disappeared).

4. **Install continuous query**. This operation allows the installation of a continuous query. Continous queries are operations which contain multiple parameters, and are essentially queries that are evaluated at specified periodicities, and whose returning values are inserted into the time series database under a specified name. This operation is useful for applications that wish to, for example, resample their data to longer periodicities, or wish to calculate a certain aggregated value at specific intervals.

5. **Broadcast messages and subscribe to broadcast message receptions**. 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 to the overlay protocol, which in turn propagates the message contents via their active connections (until the TTL is 0). Broadcast messages are propagated and forwarded to peers in the active in a similar manner to the subscription messages (described in). 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 pass through the server's overlay protocol.

quote broadcast aggregation

6. **Install Alarm**. This operation is coalesced by a subscription containing the parameters described in section , whenever a client issues this operation, the API assigns a new ID to the alarm and adds it to the alarm manager (described in section), where it begins being periodically verified. After this, if in any verification the alarm fires, the alarm manager notifies the API which in turn sends a message to the client with the firing alarm's ID.

citar seccao alarms

ref

7. **Install and removal of neighborhood aggregation set**. These operations manage the operation of the neighborhood aggregation algorithm defined in section . These are performed via requests, in case of installation, these contain contain the parameters for the request mentioned in . Whenever the api receives a request for the creation of a neighborhood aggregation, it assigns an ID to it, and sends a reply to the client with the generated ID. Conversely, when the client no longer wishes to collect these values, it issues a removal request containing originally assigned ID of the aggregation set.

put ref to neigh agg

another ref to place

8. **Install and removal of global aggregation function**. These operation initiate and stop the global aggregation procedure (described in subsection), the behavior of this operation, in regard to the interaction betweeen the client and the server, is similar to the neighborhood aggregation set. However, nodes receiving this request become roots of their own global aggregation trees.

ref section global agg

9. **Install and remove tree aggregation function**. This is the last detailed operation of the deMMOn API, which in terms of interaction between the client and the server, behaves similarly to both the neighborhood and global aggregation requests. Whenever the API receives a request to begin global aggregation, it forwards it to the aggregation protocol, which begins the mechanism described in . insert ref

falar do cliente e do exporter

4.5.1.2 Summary

In this section, we have presented an overview of the capabilities of the DeMMon API. We began by providing a brief overview of the interaction paradigm (message-based) and the reasons behind this choice. Next, we detailed the technologies used to realize this interaction paradigm and enumerated what we believe to be its most relevant operations. 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 begun 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 by 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 when possible, from a partial (or complete) set other nodes in the system. Next, we covered the monitoring module (sec. 4.4), which is the module in charge of: storing the time series data, parsing and processing queries, and managing alarm lifecycles. We finished the chapter 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 in a benchmark application for a back-end of an edge-enabled interactive multiplayer game, with functionality inspired from the popular game PokemonGO . This contribution arose from the suggestion presented in , and aims to present a materialization of a benchmark with focus on real-time interactions between users. The importance of this contribution is its' possible use in testing service deployment systems, as the user interaction can be dramatically influenced in terms of quality of service as a function of the proximity of 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).

cite

cite this https://www.semar Enabling-Novel-Edge-Enabled-Applications-Leitao-Costa/7379b13c29

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 geographical heuristics, being the latter my colleagues' work.

In this application, registered users own a set of beasts, which they can expand by catching more beasts in certain gepgraphical 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) which 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 beat, 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

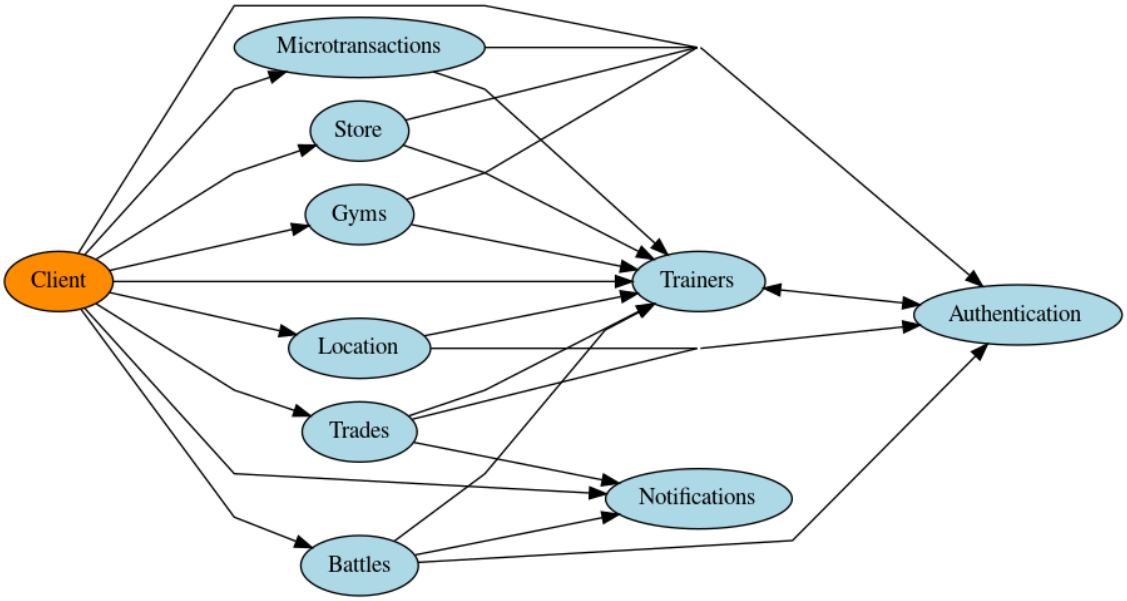

Figure 5.1: An overview of the architecture of PouchBeasts

The interactions between the services in this benchmark are illustrated in the diagram of figure 5.1. 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 regards to the recency of the information carried in the token. This service makes use of a MongoDB database to store these records in permanent storage and to maintain data consistency across microservices.

cite

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 alternatively via challenging other known users (via a notification). As previously mentioned, whenever a user is in a battle, it issues moves (based on the observed battle status) and receives updates regarding the status of the battle. As moves depend on the observed status of the battle, it is paramount (for the quality of service of the users) that both the moves of the players and the information passed from the battles/gyms service to the player suffers the least latency possible. The information regarding the issued and the status of battle is propagated to the user using WebSockets. Whenever the battle is finished, the Battles / Gyms server commits the battle result and update the users' beasts 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 which has little 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 to the services the items they wish to be traded with the other user. 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 are finished adding or removing items to be traded, they accept the trade, and the transaction result is submitted 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 backend, which it uses to propagate messages for nodes that are connected to different notification servers.

7. The last implemented microservice is called **Location** service, this service is responsible for managing the geographical locations of the users using the system, the generation and management of the generated beasts (for users to catch), and the locations of gymnasiums in a certain geographical area. Propagation of this information is done in a periodic manner via a WebSockets API. 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 . 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 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, obtained from . 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 cell centred on a user-provided location, determine the beasts and gyms to return to the 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 to, so that it does not receive only a portion of the results for its correspondent geographical area, and finally (4) to allow a dynamic subdivision and collapse of geographical regions to instantiate or decommission location servers.

cite

cite

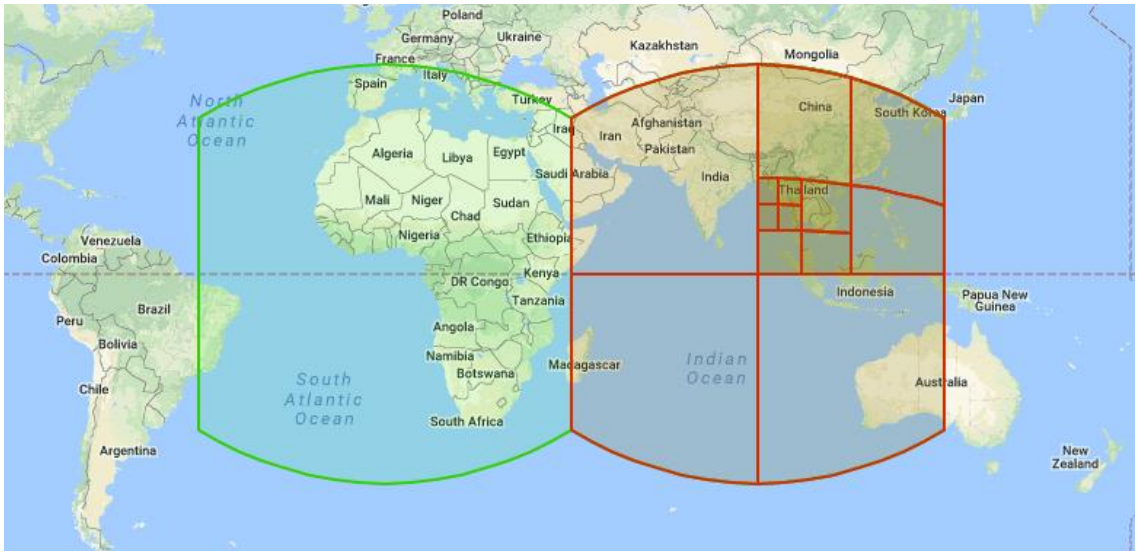

Figure 5.2: Example of S2 cell hierarchy

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 frustration when it comes to

user experience (i.e. having a delay in a battle may cause a certain users to lose in a way which feels unfair).

To enable this experimentation of 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 actions. Then, through the instrumentation of both the client and the services, we provide a way 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, users may then acess the performance of their service deployment and maintenance systems.

.

To enable automated client testing, we provide ways for client to simulate user behavior. This behavior is configurable via a configurable stochastic matrix . This matrix contains a line and a row for each possible action to perform with the client, and each matrix position (provided 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.

falar de que o design permite a utilizacao de tantos servidores quanto se quiser que fazem interacoes real-time com os utilizadores

cite this from somwhere else other than wikipedia https://en.wikiped

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) which offer a wide range of interaction types, from request-reply based interactions to real-time interactions which have 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 colleague that was contributing in the implementation of this benchmark has successfully built a system (for his own dissertation) that, through the metrics obtained by the DeMMon framework, improves the QOS of clients using the "PouchBeasts" services.

6

Evaluation

In this chapter, we test the performance and applicability of the deMMon framework. It is our goal to demonstrate the applicability of the devised solution through the comparison of multiple aspects of the framework against popular solutions (in each aspect) from the literature. To this end, section 6.1 covers the experimental setting and configuration, namely the specifications of the nodes executing the solutions, the methods used to control the experiment scenarios, among other aspects of the experimental setting. Following, in section 6.2, the applicability of the overlay protocol is tested in two ways: the first test compares the characteristics of the resulting network against a set of baseline protocols, and the second overlay test evaluates the performance of the protocol in performing message dissemination against the same set of baseline protocols (executing dissemination protocols). Lastly, section 6.3, covers the experimental evaluation for aggregation protocol, notably, which solutions composed the baseline for comparison, which experiments were carried, and provides the obtained experimental results. All of the previously mentioned sections are finished with a discussion of the obtained results.

6.1 Experimental Setting

To conduct the experimental evaluation of the devised solution, instead of resorting to simulation, solutions were implementated and tested in real-world scenarios. However: (1) as scalability is one of the components to be tested and there is a limited pool of individual machines in the testbed to conduct the experiemental evaluation; (2) in order to emulate a real-world scenario, there was the need to both limit the networking capacity and inject latency among nodes, we resorted to using containerization. Containers allowed us to to run multiple independent processes in a isolated environment, and allowed the manipulation of the networking capacity 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 machines. To solve this, we made use of docker containers, and employed a tool called docker swarm , which allows users to coordinate a set of nodes running

cite

citation

docker (denoted a swarm). Nodes in a swarm, among many other features, may perform Multi-host networking, which consists in integrating containers from different nodes into a unified network, where they are automatically assigned IP addresses and can communicate, regardless of the machine each docker is executing on. To setup the experimental scenario, we developed a set of scripts in both BASH, Python and GO to create, orchestrate, and decomission (when needed) containers such that all containers are inserted into a unified network(and assigned IPS in a pre-determined range), which are then loaded with all the necessary executables 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 limitation in a realistic way. To do so, we used data from real-world readings of real-world scenarios obtained from WonderNetwork , which is a network 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 to every other in the network (including themselves). We extracted this information, and there as there was the need to test the framework with larger node counts (up to 750 nodes), the data points were multiplied by 5 times.

https://wondernetwork.com/
meter como
cite

Then, as the obtained data from this network did not contain bandwidth information for each node, we used the country, provided by WonderNetwork, to assign bandwidth values according to the list of bandwidth per country provided by speedtest.net [21]. Provided the purpose of this framework is to perform on cloud-edge scenarios, composed by nodes inside and outside of the data-center (DC), where nodes outside the DC have lower networking capacity comparatively to nodes running inside the DC, we divided each data point by 12x (representing the nodes running outside of the DC), and divided the first N nodes by 2.5x, (corresponding to the number of nodes representing the data centers).

Provided with the networking capacity and the latency matrix for all connection pairs, there was the need to both limit the networking capacity and inject latency in the containers executing the protocols for the experiments. To achieve this, we used a tool called Traffic Control (TC) , 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. This tool sets up sets of queuing systems and mechanisms by which packets are received and transmitted, then queue (also denominated qdisc) or class specific policies decide which (and whether) packets to accept at what rate on the input of an interface and determining which packets to transmit in what order at what rate on the output of an interface.

cite

In our case, we used this tool to both limit 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 shaping of traffic according to a certain configurable rate. HTB requires programmers to set up a hierarchical class structure, where child classes, attached to a qdisc, maniulate packet order and apply certain 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, and the parent has available tokens). If the parent class is also limited, then the sum of its child classes will be limited to its rate. In our experimental configuration, each container creates two default queues (or qdiscs) attached to the inbound and outbound networking interfaces. Then, two HTBs are attached to the default inbound and outbound qdiscs (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 a rate corresponding to the configured download and outbound rate for the container 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 the intended bandwidthrate.

For each of these leaf classes, we attached a netem qdisc which applies a delay to each packet corresponding to the latency between the origin and the target container. In our experimental setup, packets are forwarded using filters, in the case of the measurement traffic the filtering was performed via installing a high-priority filter verifying the source and destination ports of the packets and sending it to the measurement classes. The remaining traffic was then forwarded to the default traffic class. The Routing from these two outbound classes to the leaf classes is done via filtering the destination IP address of the packets. The objective of separating the traffic in these two classes 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.

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) cite The hardware from this testbed used to carry the experiments was sets of 10 physical nodes for experiments with 50 and 250 logical nodes, and sets of 30 physycal nodes for experiments with both 500 and 750 logical nodes. Each physycal machine 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 results 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 in the overlay protocols' evaluation.

6.2 Overlay Protocol: Experimental Evaluation

In this section, we present the results obtained from 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 establised networks' efficiency (according to latency); and (3) finally, testing demmons' message dissemination capabilities against the same baseline set of baseline protocols, executing two distinct dissemination protocols. 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 (discussed in further detail in section 2.3) to perform the overlay protocol network establishment and construction comparison were: (1) **Hyparview** , which is a protocol that builds a non-structured overlay network using a fixed-sized view materialized by active bidirectional TCP connections (these connections also used for fault tolerance); the second baseline protocol is **X-Bot** , which is a protocol that essentially employs Hyparview to establish the initial overlay structure, and optimizes (whenever possible) the overlay network according to a configurable heuristic. These optimizations are performed via gossip mechanisms and improve active connections' costs. It is important to mention optimizations are performed in such a way that maintains the guaranteed established by the Hyparview protocol, through a two-step exchange of the active connections of two distinct pairs of nodes, in a coordinated manner. The third implemented baseline protocol was **Cyclon** , which is an overlay protocol that materializes a network composed by asymetric links via periodic exchanges of node pointers with a configurable age. The last implemented baseline was **T-Man** , which is a protocol that iteratively builds on an existing set of nodes to build a new, 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 view for this overlay protocol, we employed the Cyclon protocol, which is why in the evaluation results for this protocol are labeled as "Cyclon T-Man". All of the described baseline protocols were, for comparativeness, implemented using the GO-BABEL framework (described in section 3.2), and, as GO-BABEL only provides unidirectional connections, protocols which

Table 6.1: Protocol test 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	-	

require bidirectional connections (hyparview, x-Bot and deMMon) were added periodic mechanisms to ensure that the two connections were established.

The utilized parameters for the protocols were adjusted to attempt to perform a fair comparison of the evaluated protocols, which are displayed in table 6.1, the first column called "VSizeMax" represents the maximum size of the active view, which in most protocols was set as 5 except for cyclon where it was set as 7 as it is the only protocol without a backup secondary view. In the case of deMMon, this value represents the maximum number of children per node. The second parameter, denominated "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 as 25 for deMMon, hyparview and X-BOT, and set as 7 for the case of Cyclon T-Man (which corresponds to the size of the cyclon view, running in the background, to feed its initial view). The next parameter, called "Shuffle", corresponds to each protocols' shuffle mechanism, which is configured to execute every 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 parameter worth mentioning 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 dialing nodes, receiving message responses, among others) are lower than 5 seconds, and all periodic mechanisms are executed with a frequency lower than 15 seconds.

pode haver mais parametros / talvez meter uma tabela completa em anexo?

6.2.2 Overlay construction and maintenance: experimental results

The first conducted experiment, aimed at evaluating how protocols build and maintain the overlay networks, is an experiment where different numbers of nodes join the system and remain during 25 minutes. In this experiment, we evaluate what are the properties of the built overlay networks (costs, degree distribution, among other properties) and how fast the protocol converges towards an optimized network. Finally, in order to compare the scalability, performance and fault-tolerance at multiple scales, we perform the previously mentioned experiment using network sizes of 50, 250, 500 and 750 nodes.

In the 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 an experiment with no failures. For both of these graphs, we show the results obtained

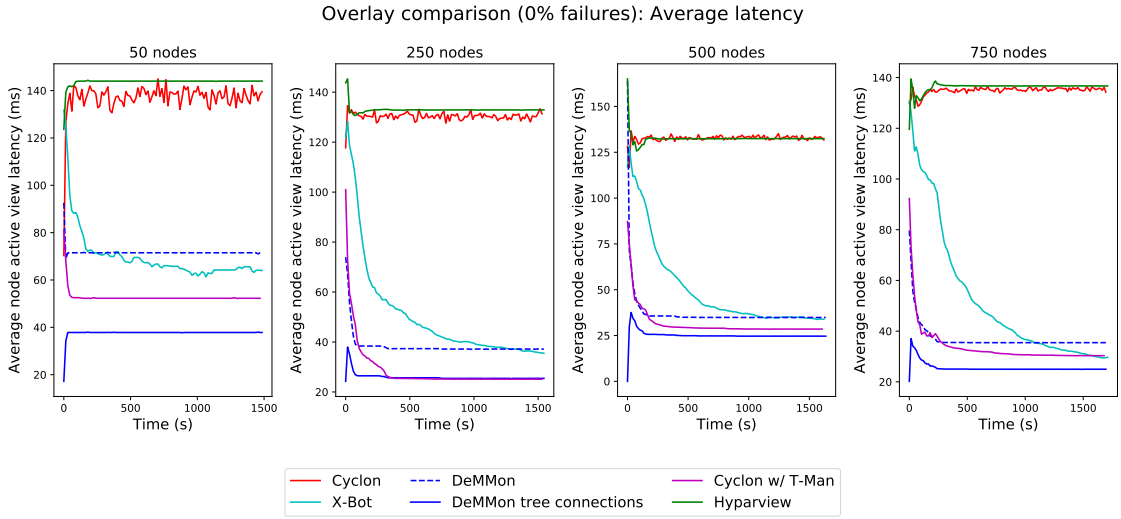

Figure 6.1: Average latency in per node in established networks

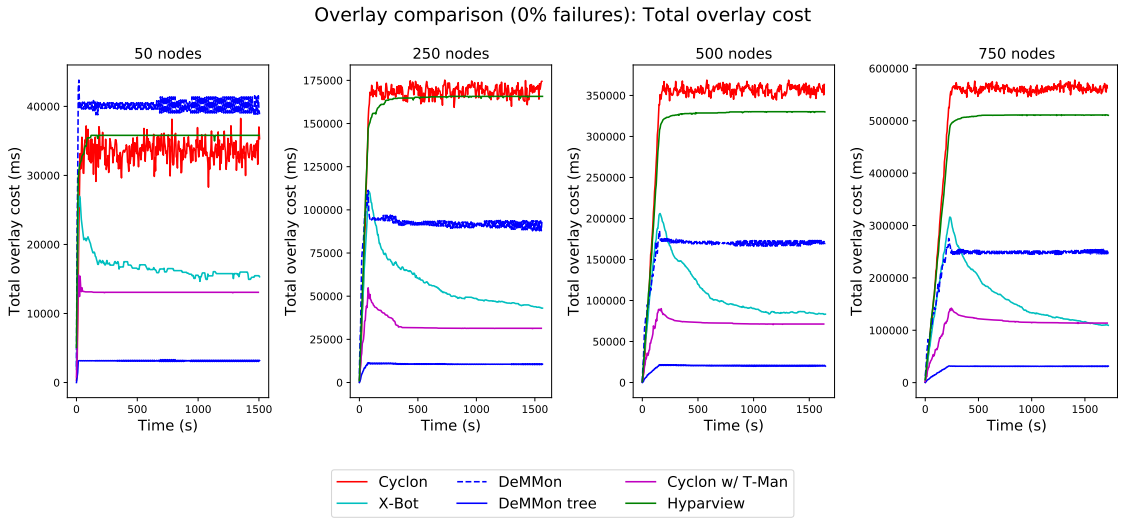

Figure 6.2: Total network cost (in latency)

from both the baseline protocols and the deMMon protocol. In the case of deMMon, we make a distinction between twos latency values, 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 (i.e. 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 correllation from improving only the parent connections to the sibling latencies. The second reason to make this distinction is due to the fact that these connections are significantly more used when compared with the sibling connections for network maintenance, information dissemination and in-transit aggregation.

The results displayed in graphs 6.1 and 6.2 show that both hyparview and cyclon average their latencies at around the same values (which also correspond to the average of all connections of the latency matrix), which is expected as these protocols do not perform optimizations in regard to the network latency. The results also show that the devised protocol is the fastest in regard to converging to its lowest latency value, and that X-Bot is the slowest, not converging to a final value in a test of 25 minutes, which is expected as X-Bots' overlay improvements are performed using 7 messages, contrasting heavily with deMMons' 2 required messages, and T-Mans' 0 required messages for performing overlay improvements. While the total and average latency of the deMMon overlay is not the lowest in any of the displayed results, when comparing only the vertical connections, deMMon reaches a total latency cost lower than any other tested procotocol. This is important because 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 mean latency in the least amount of time, it does so disregarding the fact that nodes may become disconnected from overlay, which as we will observe further in this chapter, prevents this protocol from being a basis for reliable message dissemination. This factor 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 for each tested protocol (these results correspond to the last configuration of the network before the experiment ended). These results show that T-Man, at multiple node counts, posesses 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 results 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

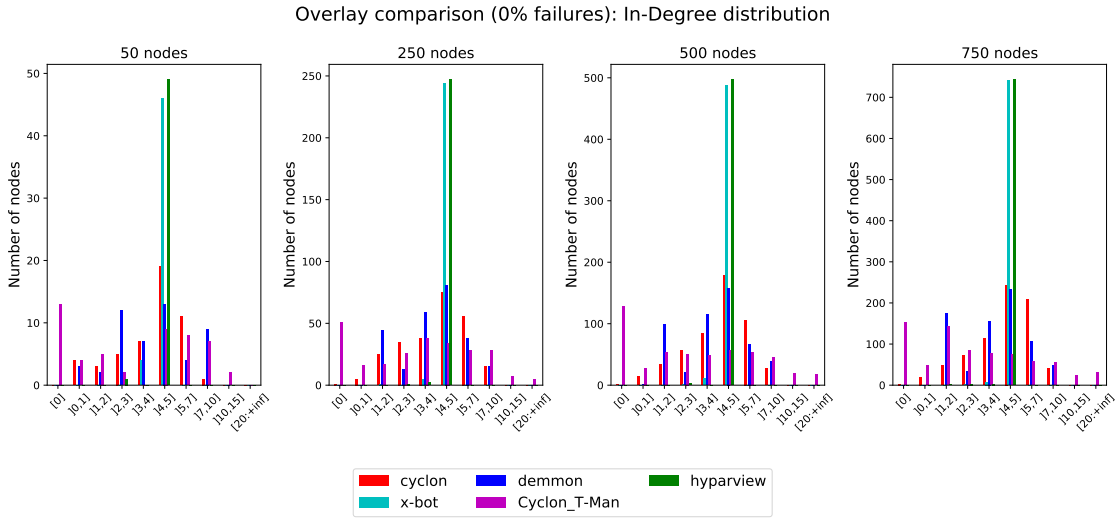

Figure 6.3: Node in-degree

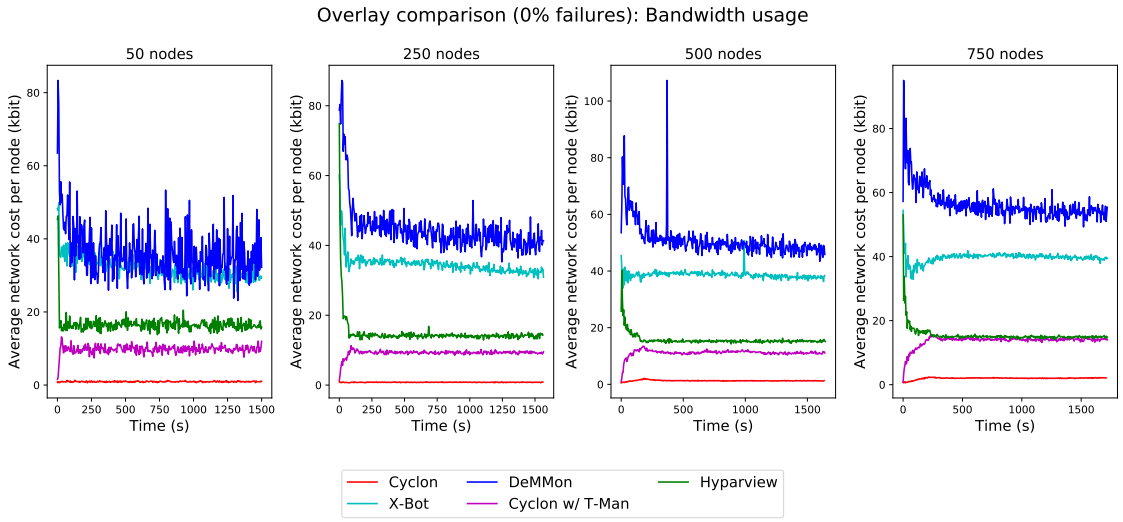

Figure 6.4: Protocol bandwidth cost

number of children of 5 (which as previously mentioned, is not guaranteed to bound the number of children).

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 due to the fact that deMMon exchanges more information periodically with peers in the active view (to maintain an improve the tree structure) when compared to the other protocols. Conversely, the protocol which uses the least amount of bandwidth is Cyclon, as its shuffle mechanism

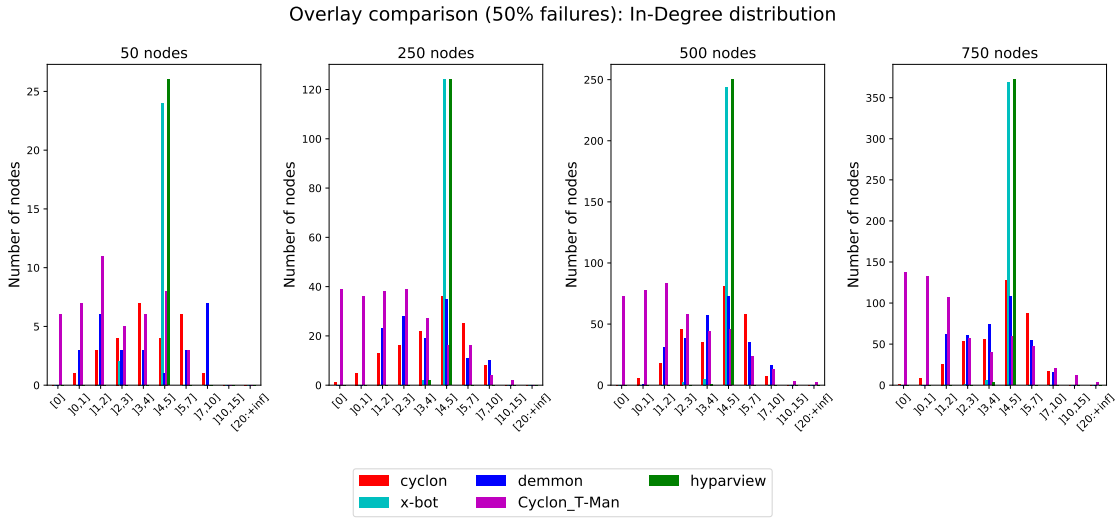

Figure 6.5: Node in-degree (50% failures)

is relatively inexpensive and the protocol posesses no other mechanisms. Although protocols have varied networking costs, we believe that even deMMon which uses more 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 the results for the in-degree distribution of the protocol in a scenario with failures. This second experiment tests the fault tolerance of the protocols by first establishing the network (similarly to the first scenario), however, during the middle of the experiment, we induce a catastrophic failure of 50% of the nodes. The objective of this experiment was to test if any node became isolated from the network after this period of 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 the devised protocol can recover from faults effectively.

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

meter resultados grandes em anexo?

6.2.3 Information dissemination: experimental results

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 havent done so previously (excluding the sender). The second used dissemination protocol was PlumTree , which is a dissemination protocol that builds a dissemination tree rooted on the first node that issues a broadcast message. The reasoning behind this choice of dissemination protocols was to provide a fairer comparison of deMMon with the remaining protocols, as we believe that because simple flood generates many redundant messages when compared to dissemination using only a tree structure, it would be unfair to not include a dissemination protocol which also employs a tree, similarly to deMMon. It is important to mention that, when testing the PlumTree protocol,in order to establish the initial tree, in the experiments for this protocol a single node first establishes the tree by starting the dissemination of its messages a minute earlier. 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).

cite

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 node numbers we also tested failure rates of 0 and 50% of the system. For each of these node numbers and failure rates, 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 percentage of nodes participating in the overlay at the time of emission of a certain message that receive that message; (2) the maximum message throughput reached by every protocol in a 30 second window, (3) the average latency taken by messages until they reach their destination, and (4) the bandwidth usage of each of the protocols.

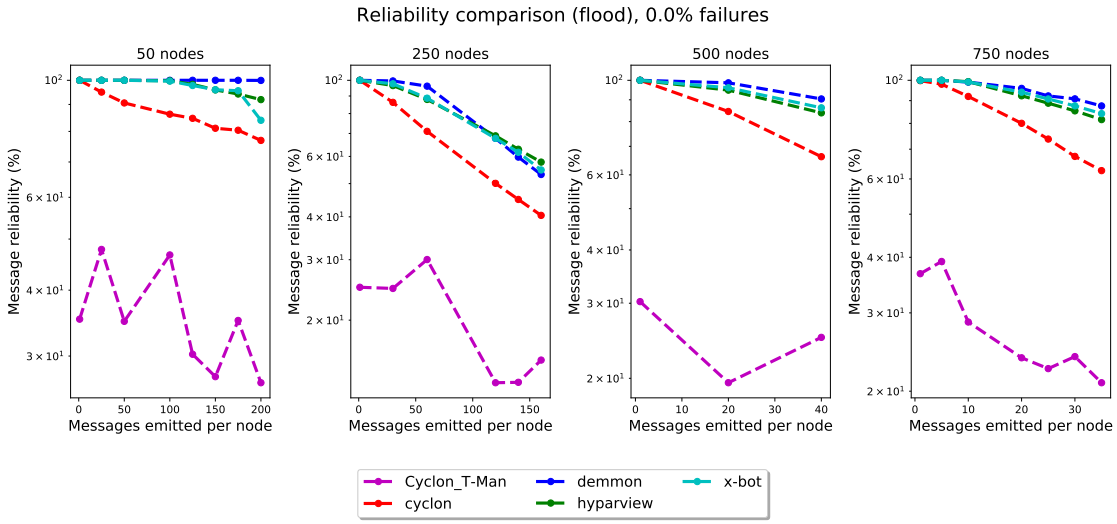

Figure 6.6: Average message reliability in simple flood scenario (0% failures)

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

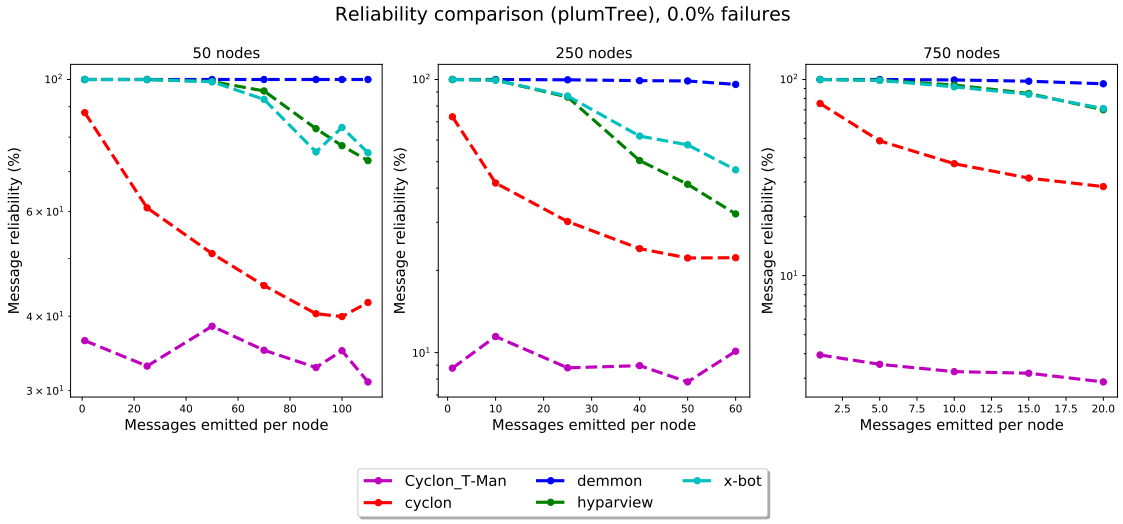

Figure 6.7: Average message reliability in PlumTree scenario (0% failures)

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 is exceeded. Whenever this occurs, as certain messages are 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 there are multiple nodes emitting new messages simultaneously, and new nodes can become saturated while performing this mechanism, the tree structure may never reconverge until all messages are delivered. Until this occurs, the protocol essentially becomes a push-pull gossip protocol, which has lower performance in our experiments in terms of reliability, because the tests end before the protocol has delivered all messages.

In addition to the previously mentioned reasons, in an ocasions where a node has received an IHAVE message for a certain message ID, and happens to have available upload bandwidth but its download capacity is all taken up by incomming traffic, this node will periodically emit GRAFT messages to the sender of the IHAVE nessage, which will reply with broadcast messages that are only received after a large time frame. Whenever this occurs, there are 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, given that the tree is not defined by the propagations of broadcast messages, it 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 incomming connections which do not receive any broadcast messages from other nodes, and in the case of cyclon we believe the drop in reliability is attributed to the use of UDP as a its communication medium, which means that whenever the data channels becomes saturated, many of the broadcast messages are lost, contrary to deMMon, hyparview and X-BOT, that use TCP and do not drop messages in congestion periods. Finally, we believe that both cyclon and T-MAN when paired with plumTree have particularly lower reliability for 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 smaller, as when the tree heigh 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 this. 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.

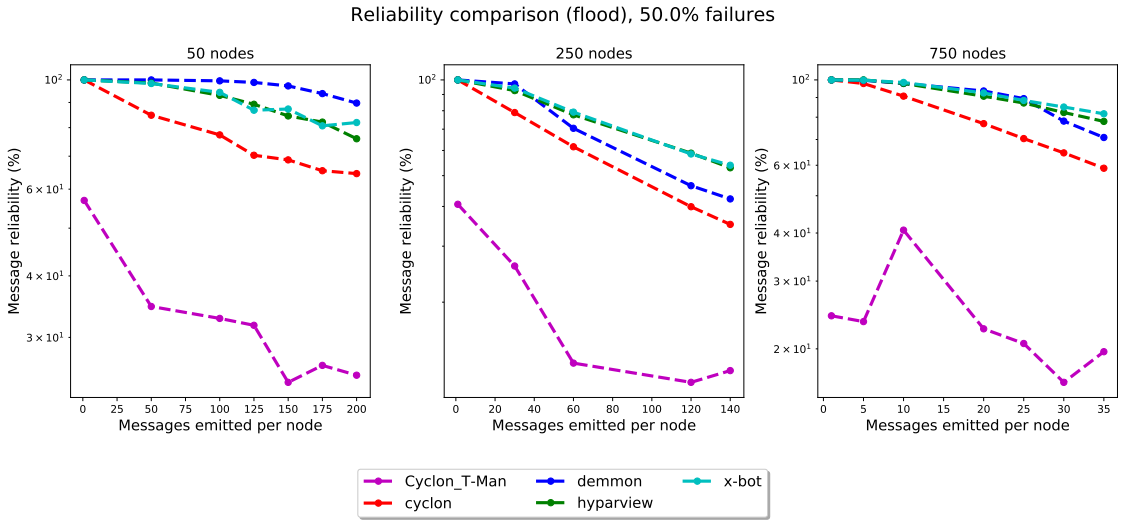

Figure 6.8: Average message reliability in simple flood scenario (50% failures)

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 and lose their parent, the failure recovery mechanisms may take a long time frame to execute, and during this period

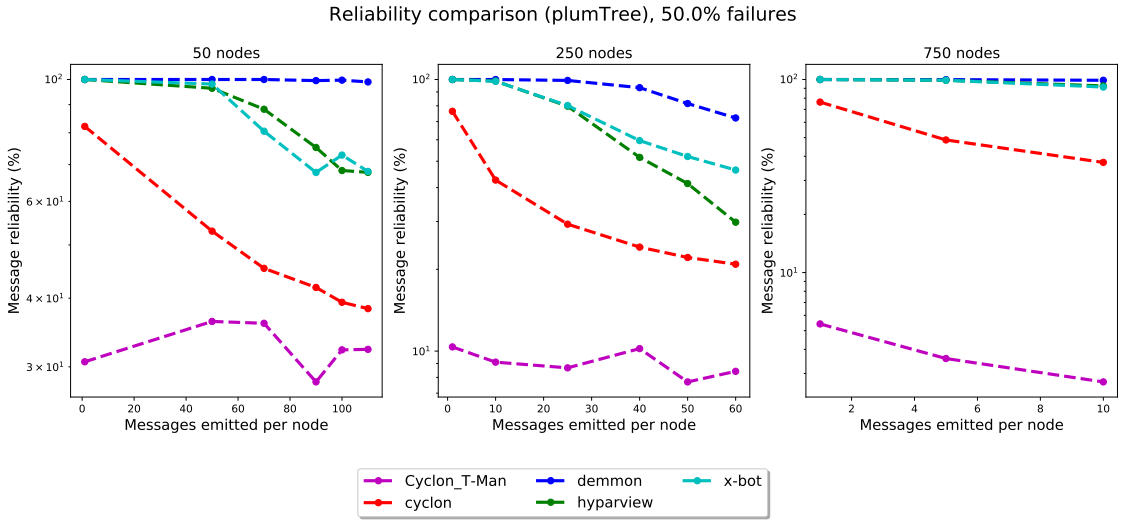

Figure 6.9: Average message reliability in PlumTree scenario (50% failures)

nodes are disconnected from the remaining overlay and consequently do not receive or send message to 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 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, all obtained results are available in annex .

put the results in annex, and ref

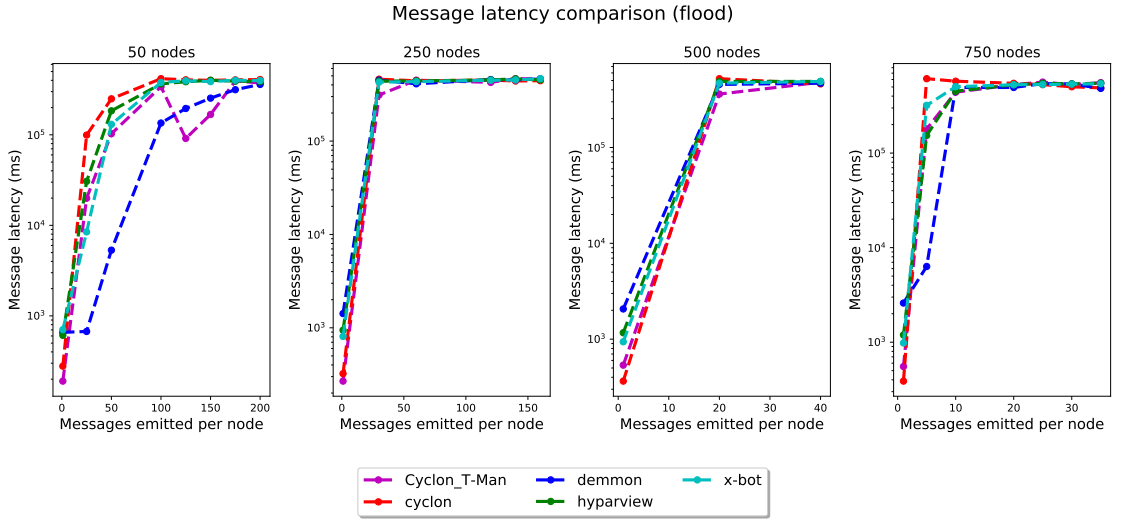

Figure 6.10: Average message latency (in ms) in simple flood scenario (0% failures)

In figure 6.10 we may observe the obtained results from collecting the latency between

the emition and reception of the message for each node. The first takeaway from these results is that all protocols plateau at the same latency value, 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, and 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 flood protocol becomes saturated due to the number of redundant messages sent, whereas deMMon must send less messages to perform the message dissemination.

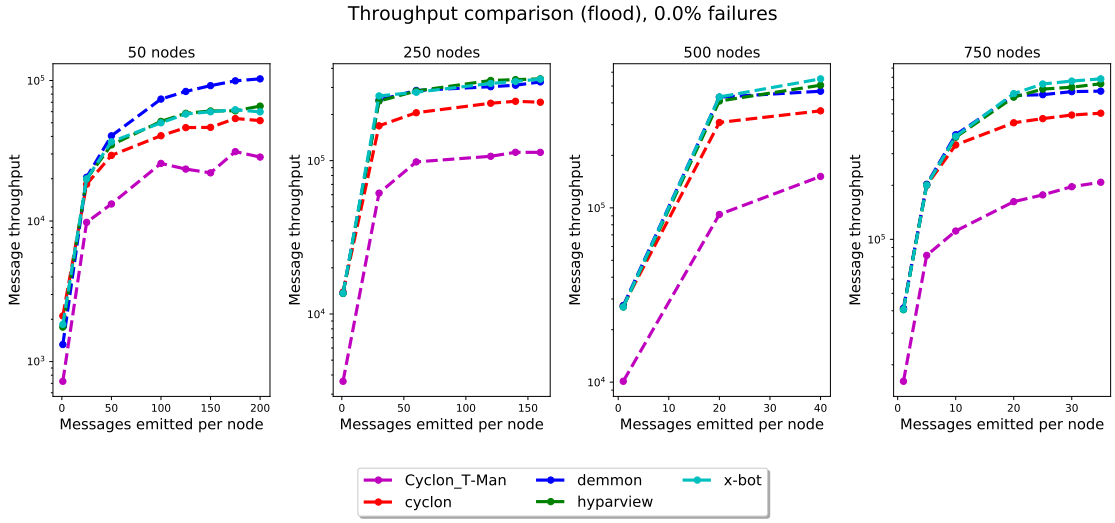

Figure 6.11: Maximum message throughput during experiment (30 second window) in simple flood scenario (0% failures)

In figure 6.11, we may observe the obtained throughput across the simple flood experiments with 0 failures, as we can observe, in lower node counts, the throughput achieved by deMMon surpasses the throughput achieved by the remaining protocols on lower node counts (i.e. 50 nodes), 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 1 message emitted per node. In the left we may observe the results obtained by the execution of PlumTree with the baseline protocols, while in the right we

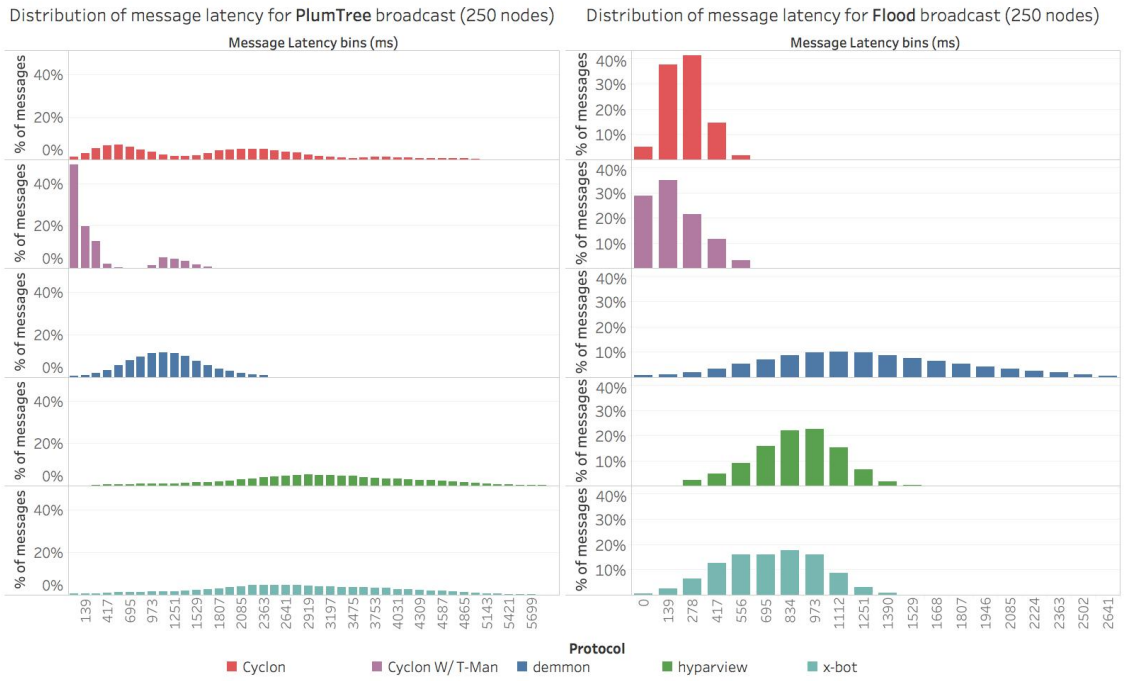

Figure 6.12: Message latency distribution in scenario with low network saturation

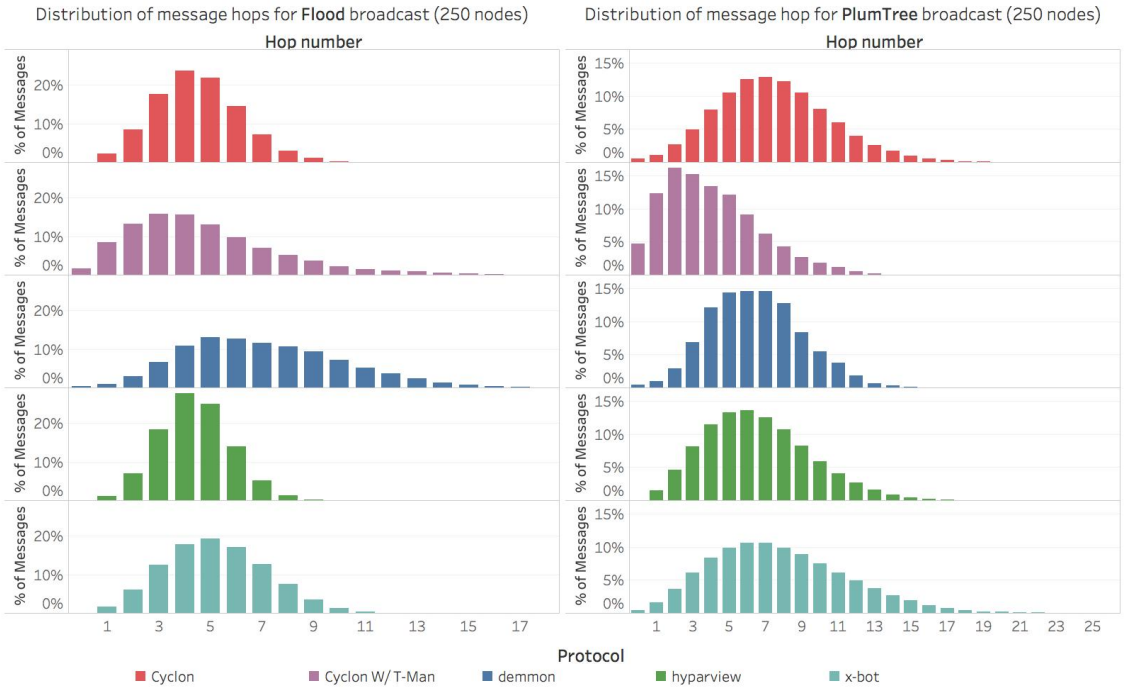

Figure 6.13: Message hop distribution in scenario with low network saturation

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 be explained by the fact that, by employing a single shared tree to disseminate the messages, as the messages must take specific routes in the broadcast procedure in order to decrease message redundancy, messages have to take more hops to get to their destination, consequently achieving 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 1 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 observable in the graphs displayed in 6.6 and 6.7.

6.2.4 Summary

In this section we covered the obtained results from 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 was the ability to establish and maintain the overlay connections, where results show that the devised protocol is consistently the fastest protocol 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 lower 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 scales its throughput tends to plateau at around the values as both X-Bot and Hyparview when paired with 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 the simple flood alternative, which is caused by messages taking more hops to reach their destination.

To conclude, we believe to have built an overlay protocol that performs competitively with popular solutions from the state of the art for performing information dissemination. The conducted tests suggest that DeMMon performs particularly better for 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), performs worse when compared to simple flood protocols.

6.3 Aggregation Protocol: Experimental Evaluation

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 monitoting solution from the state of the art, named Prometheus . We begin by providing cite 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 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.

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 will provide the obtained results regarding the applicability of each of these features, however, as Prometheus does not provide a comparable feature to the implemented neighborhood aggregation feature, this feature will be tested in an isolated manner. 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 "supposed" value and $aggVal$ corresponds to the obtained aggregated value during the experiment. In addition to the error over time, we collected other metrics to acess 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.

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 node collects and aggregates the metrics correspondent to all the nodes in the system, collecting a global view of the system (materializing a single-level tree). The second experimental setup, named **Prometheus tree**, corresponds to a more sophisticated setup where instead of having a single aggregating node, a portion of nodes in the system aggregate the metric values (effectively splitting the load among the aggregator nodes), these aggregator nodes, similarly to deMMon, are set up in the shape of a tree, and make use of federation to scrape the partially aggregated value from other prometheus. The generation of these configurations is performed in an automated manner through scripts. 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, 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 them to study the impact on the network cost of performing in-transit aggregation by every node when compared to performing the aggregation of metrics corresponding to multiple nodes on a single node.

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 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 to maintain the desired ratio.

6.3.2 Global aggregation

7

Conlusions and future work

Bibliography

[1] 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).

[2] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (Sept. 2014), pp. 81–84. ISSN: 2372-2568. DOI: 10.1109/MCC.2014.51 (cit. on pp. 9, 10).

[3] F. Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16 (cit. on p. 7).

[4] 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. 19).

[5] Cloudviz. *cloudviz/agentless-system-crawler*. July 2019. URL: https://github.com/cloudviz/agentless-system-crawler (cit. on p. 21).

[6] 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. 18).

[7] 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. 22, 23).

[8] 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. 19).

[9] G. DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220 (cit. on p. 20).

[10] *docker stats*. Jan. 2020. URL: https://docs.docker.com/engine/reference/commandline/stats/ (cit. on p. 21).

[11] 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. 16).

[12] *Empowering App Development for Developers*. URL: https://www.docker.com/ (cit. on pp. 9, 21).

[13] 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).

[14] 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. 19).

[15] Google. *google/cadvisor*. Jan. 2020. URL: https://github.com/google/cadvisor (cit. on p. 21).

[16] 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).

[17] *Gtk-Gnutella*. Dec. 2019. URL: https://sourceforge.net/projects/%20gtk-gnutella/ (cit. on p. 19).

[18] 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. 16).

[19] 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 p. 2).

[20] *Infrastructure for container projects*. URL: https://linuxcontainers.org/ (cit. on p. 9).

[21] *Internet Speed around the world*. URL: https://www.speedtest.net/global-index#mobile (cit. on p. 72).

[22] 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 p. 13).

[23] 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. 23).

[24] M. Jelasity et al. "Gossip-based peer sampling". In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es (cit. on p. 11).

[25] 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 p. 22).

[26] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40 (cit. on p. 20).

[27] 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. 12, 14, 21).

[28] 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. 12, 14).

[29] 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. 21).

[30] 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, 7).

[31] 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 p. 14).

[32] 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. 12, 15).

[33] 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).

[34] 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. 15).

[35] J. Liang et al. "MON: On-Demand Overlays for Distributed System Management." In: *WORLDS*. Vol. 5. 2005, pp. 13–18 (cit. on p. 14).

[36] Y. Mao et al. "A Survey on Mobile Edge Computing: The Communication Perspective". In: *IEEE Communications Surveys & Tutorials* PP (Aug. 2017), pp. 1–1. DOI: 10.1109/COMST.2017.2745201 (cit. on p. 7).

[37] 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. 24).

[38] 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. 12, 16).

[39] 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. 12, 16).

[40] G. Peng. "CDN: Content distribution network". In: *arXiv preprint cs/0411069* (2004) (cit. on p. 7).

[41] E. Preeth et al. "Evaluation of Docker containers based on hardware utilization". In: *2015 International Conference on Control Communication & Computing India (ICCC)*. IEEE. 2015, pp. 697–700 (cit. on p. 10).

[42] Prometheus. *From metrics to insight*. URL: https://prometheus.io/ (cit. on p. 25).

[43] 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. 12, 23, 25, 26).

[44] 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. 15, 16, 25).

[45] 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. 16).

[46] 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 p. 2).

[47] *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).

[48] 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 pp. 1, 6).

[49] J. E. Smith and Ravi Nair. "The architecture of virtual machines". In: *Computer* 38.5 (May 2005), pp. 32–38. ISSN: 1558-0814. DOI: 10.1109/MC.2005.173 (cit. on p. 9).

[50] 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. 12, 15).

[51] 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. 10).

[52] 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. 19).

[53] "Topology Management for Unstructured Overlay Networks." In: *Technical University of Lisbon* (2012) (cit. on pp. 11, 13, 17, 18, 20).

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

[55] T. Verbelen et al. "Cloudlets: Bringing the Cloud to the Mobile User". In: *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*. MCS '12. Low Wood Bay, Lake District, UK: Association for Computing Machinery, 2012, pp. 29–36. ISBN: 9781450313193. DOI: 10.1145/2307849.2307858. URL: https://doi.org/10.1145/2307849.2307858 (cit. on p. 6).

[56] M. Villari et al. "Osmotic computing: A new paradigm for edge/cloud integration". In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83 (cit. on p. 7).

[57] 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. 25, 26).

[58] 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. 16).