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 proposto pelo paradigma da Computação na Nuvem diverge do modelo das aplicações para a Internet das Coisas e para aplicações móveis, dado que a maioria da produção e requisição de dados é feita por dispositivos que se encontram distantes dos centros de dados. Armazenar dados e executar computações predominantemente em centros de dados incorre em custos de infrastrutura adicionais, aumenta a latência para os utilizadores e para fornecedores de serviços, como também levanta questões sobre a privacidade e segurança dos dados.

Para mitigar as limitações previamente mencionadas, surgiu um novo paradigma: Computação na Periferia. Este paradigma propõe executar computações, e potencialmente armazenar dados, em dispositivos fora dos centros de dados. No entanto, à medida que nos distanciamos dos centros de dados, a capacidade de computação e armazenamento dos dispositivos tende a ser limitada. Tendo isto, surge a necessidade de partilhar recursos entre dispositivos na periferia, de modo a executar computações sofisticadas que outrora seriam impossíveis com um único dispositivo destes.

O estudo do estado da arte revelou que as soluções existentes para a gestão e localização de recursos são normalmente especializadas para ambientes na Nuvem, onde os dispositivos têm capacidade de computação e armazenamento semelhantes, algo que não é adequado para ambientes dinâmicos e heterogéneos como a periferia do sistema. Nesta dissertação, propõe-se a criação de uma solução para a gestão e monitorização de recursos na periferia. Esta solução não só pretende gerir grandes quantidades de dispositivos, como também recolher e agregar métricas sobre a operação e execução de componentes aplicacionais, de forma descentralizada. Estas métricas, por sua vez, auxiliam a tomada de decisão relativa à migração, replicação ou delegação (de porções) dos componentes aplicacionais, permitindo assim a adaptação autonómica do sistema.

Palavras-chave:
Computação na periferia, Gestão de recursos, Monitorização, Localização de recursos, 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 [32]. 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 [29] [47]. 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 [46], 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 [53], Omega [45], 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

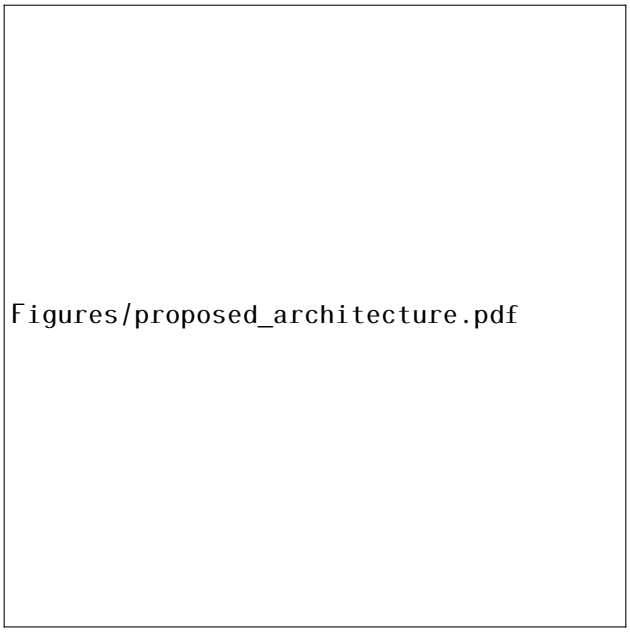

Figure 2.1: High-level architecture for a resource sharing platform

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 [29]. It has the potential of enabling novel edge-enabled applications along with optimizing existing systems [47], making them more responsive.

Many approaches have already leveraged on some form of Edge computing in the past. **Cloudlets** [54] 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 [39] 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 [55] 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 [35] (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. [29], 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 [48].

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 [40], 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.

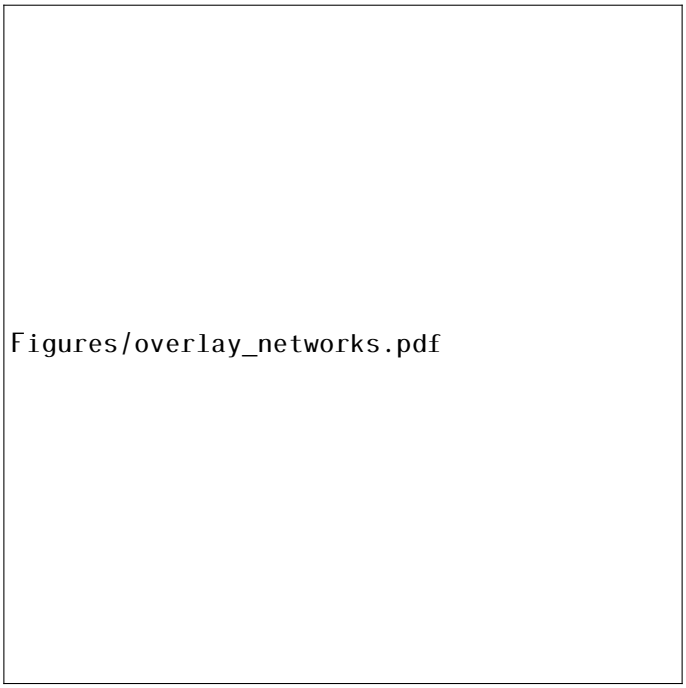

Figure 2.2: Examples Overlay Networks

However, as the system scales to larger numbers of peers, concurrently entering and leaving the system (a phenomenon called churn [50]), 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 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 [52]. 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 [23]), 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 [26] 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 [31]). 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 [27]) 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 [42])). However, trees are very fragile in the presence of faults [27].

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 [49]), however, not all distributed hash tables rely on rings to perform effective routing. For example, in Kademlia [37], 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 [38]). 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 [52]:

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 [21] 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 [34] (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 [26] (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 [27]). 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 [30] 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 [26]. 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 [31] 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 [49] 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 [33], 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 [43] 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 [44] 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 [57] Is a DHT similar to Pastry [43], 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 [37] 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 [43]) 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 [38] 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 [52], 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 [52]: **(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** [52].

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 [51], 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**) [52].

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 [25] 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 [28] proposes a decentralized device monitoring system by employing Hyparview [26] 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] [24]. 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.

	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

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 [24].

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 [22], 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 [42] 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 [36] 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 [56] (Scalable Distributed Information Management System) proposes a combination of techniques employed in Astrolabe [42] and distributed hash tables (in this case, Pastry [43]). 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 [41] 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 [56] 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 [42] 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.

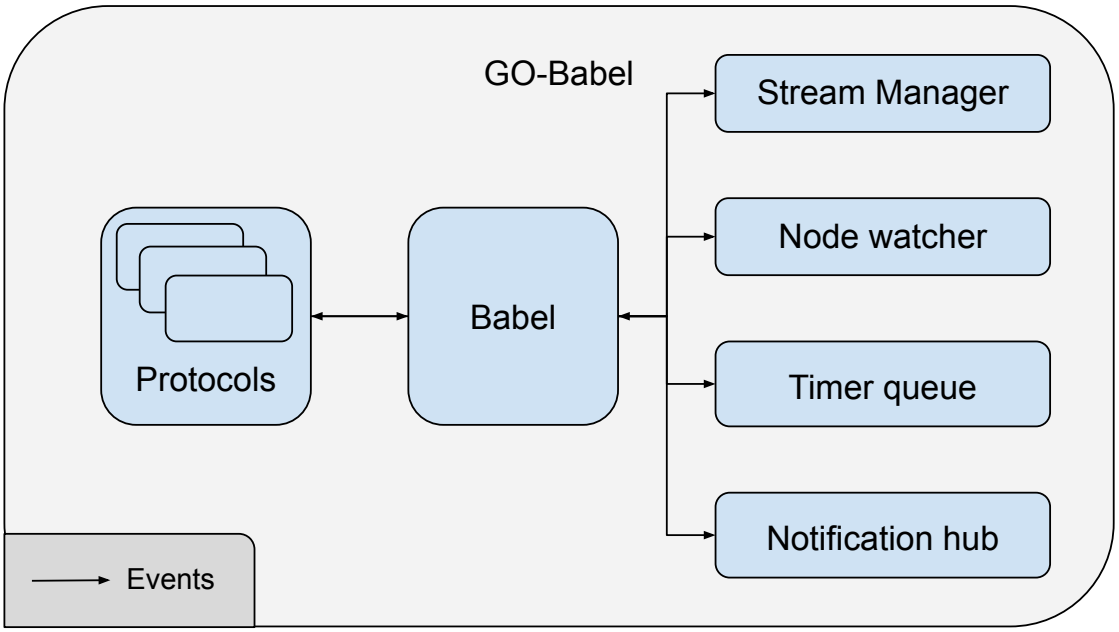

Figure 3.1: An overview of the architecture of GO-Babel

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 fulfil these requirements.

4.1 Framework overview

The devised framework, (ilustrated in figure 4.1), is composed of four main modules, the overlay network, the aggregation protocol, the API, and the monitoring module. We now describe each module's role within the framework and how they contribute to fulfil 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, the API's purpose is to expose the results and mediate the interactions between the clients and the remaining modules.

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

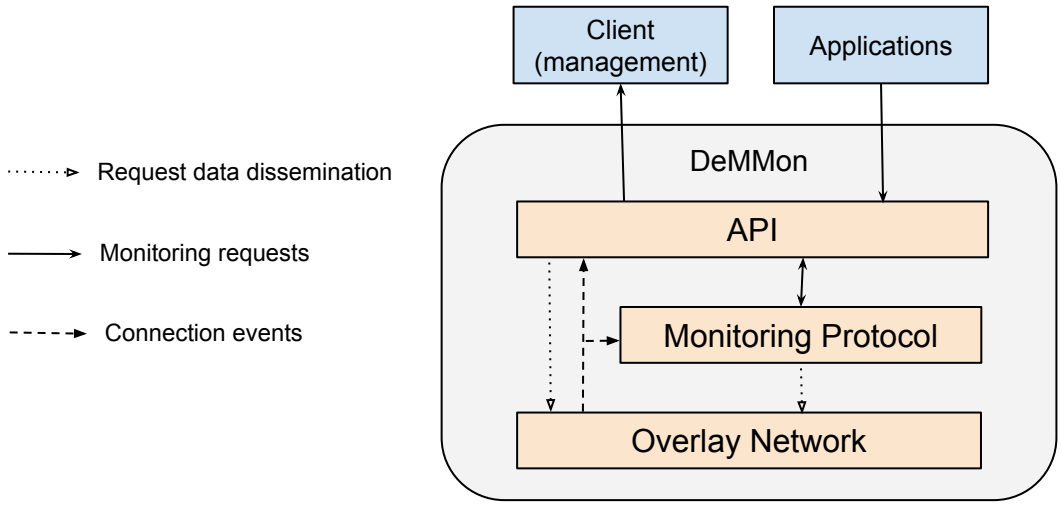

Figure 4.1: An overview of the architecture of DeMMon

their location either from one tree to another or within their tree until they have an optimized set of nodes (according to latency). The connections resulting from the operation of this protocol are the basis for the aggregation protocol. In addition, this module also offers limited 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 tree, collecting a globally aggregated value in the tree roots, or a partial view of the system for nodes which 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

criar imagem para ilustrar estrutura resultante

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**.

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.

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

Algorithm 1 Join Protocol

```
1: Types
2:     Node : <lat, parentIP, nrChildren, replied, IP, ID, coords, version, children<IP, nrChildren
3:
4:
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:
```

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).

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

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:
```

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

4.2.2.3 Passive view maintenance

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 view to the sample (discarding nodes already in the view 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).

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:
```

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.

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

With a membership protocol capable of coordinating nodes into building a tree overlay network, a new range of options open for both the dissemination and aggregation of information. In this section, we discuss the devised decentralized aggregation protocol, which leverages on the tree structure to enable the on-demand collection of information (or metrics) in a decentralized manner. While in this work we present the protocol leveraging the overlay protocol defined in 4.2, it is important to notice that the protocol is agnostic to which overlay protocol is executing, as long as it has the following characteristics: (1) it forms one or more network-shaped trees whose roots are interconnected, nodes in this tree must be connected to their parents, children and siblings (active view) in a bidirectional manner, and must provide callbacks for each node that is added or removed from the nodes's active view.

This protocol provides three different types of decentralized aggregation techniques, inspired from the study of the state of the art: (1) tree aggregation, (2) neighbourhood aggregation, and finally, (3) global aggregation, which we now clarify in further detail, starting by tree aggregation.

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**). It is important to notice that 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, the descendant node, will, when possible, embed its tree into the ascendants' (thus not performing the mechanism individually for nodes with overlapping trees, and reusing the values of the already existing tree).

The pseudocode for this mechanism (as defined in 4) begins by defining the necessary state (line 1) for its execution, starting by the active view, composed of the parent, children, and siblings of the node, this state is maintained by the overlay protocol and changes to it are propagated through notifications (this process is omitted from the pseudocode). Then, it declares three maps, the first contains the necessary metadata for each aggregation tree, values of this map contain: (1) the height of the tree, (2) the merge

function, (3) the query to generate local values, (4) the periodicity to export values (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).

Algorithm 4 Tree aggregation

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

Tree aggregation is started by the API sending a request to the protocol signalling the node should begin the procedure (line 8), this request contains: the maximum height of the tree, the merge function, the query to obtain the local value, the periodicity to execute the mechanism, and lastly, the resulting metric name. When a node receives this request, it generates a tree ID by hashing the concatenation of the tree height, the merge function, the query, the mechanism periodicity and finally the resulting metric name. This hashing is done so aggregation trees with different roots are embedded into each another, for example, if a certain node wishes to create an aggregation tree with height 5 rooted on itself, then, if a children of his (1 level lower) also creates an aggregation tree with height 4, the two trees will be embedded (as their ids will match), meaning the parent's aggregation results will be used to feed the locally requested values. It is important to notice that if the tree height is -1, then it is treated as if it had infinite height.

Before adding the tree to its local tree map, the node checks if there already is a tree with the same ID (meaning it has the same tree height, merge function, query, periodicity and resulting metric name) in the "tIds" map, if there is, then it sets the flag signalling the value should be exported locally, otherwise it adds a new entry to the map and sets up a periodic timer for that aggregation tree (lines 9 to 16).

Whenever this timer triggers, (line 18), the node checks if the aggregation tree has expired (i.e. if the parent stopped refreshing the aggregation tree), if it it has, then the node cancels the timer and deletes the tree metadata, if it has not expired, then the node evaluates the query (this procedure will be explained in section 4.4), obtaining its local value, then it merges all the values sent by its children with its own using the merge function, producing the final aggregated result. Afterwards, if configured to do so, it will store the value locally, and if the parent is also subscribed to the aggregation tree, it propagates the obtained value to it. Whenever a node receives this propagation of values from another node (line 32), it checks that it has the corresponding aggregation tree in its local set of trees and that the message was sent by its children, discarding it if it is not, and stores the received values into a map of maps, containing all children values for each aggregation tree.

Finally, each node periodically (using configured intervals) broadcasts to its children a message containing the aggregation trees it is the root of and that have height higher than 0 (line 38). Whenever this message is received (line 47), the child merges all the received aggregation trees with registered ones, marking the sender as a subscriber to values for each tree which were previously present, and propagates the aggregation trees in which he is not the root to its children.

talvez meter uma conclusão aqui?

4.3.2 Neighborhood aggregation

Neighborhood aggregation is the mechanism responsible for collecting metrics

The second mechanism

4.3.3 Global aggregation

4.3.4 Summary

4.4 Monitoring module

4.4.1 Overview

4.5 API

4.5.1 Overview

4.6 Showcase

5

Benchmark

6

Evaluation

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 p. 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. 22).

[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. 22).

[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. 10, 22).

[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. 22).

[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. 17).

[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. 10).

[21] 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. 14).

[22] 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. 24).

[23] M. Jelasity et al. "Gossip-based peer sampling". In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es (cit. on p. 12).

[24] P. Jesus, C. Baquero, and P. S. Almeida. "A Survey of Distributed Data Aggregation Algorithms". In: *CoRR* abs/1110.0725 (2011). arXiv: 1110.0725. URL: http://arxiv.org/abs/1110.0725 (cit. on pp. 22, 23).

[25] 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).

[26] 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, 15, 21).

[27] 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).

[28] 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).

[29] 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, 7, 8).

[30] 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. 15).

[31] 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).

[32] 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).

[33] 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. 16).

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

[35] 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).

[36] 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. 25).

[37] 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. 13, 16).

[38] 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. 13, 17).

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

[40] 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).

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

[42] 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, 24–26).

[43] 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. 16, 25).

[44] 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).

[45] 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).

[46] *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).

[47] 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, 7).

[48] 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).

[49] 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. 13, 15).

[50] 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. 11).

[51] 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).

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

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

[54] 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. 7).

[55] 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).

[56] 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).

[57] 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).