⟨**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 [**10.1145/1721654.1721672**]. 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 [**10.1145/3154815**]. All of these application domains are characterized by having data being generated and accessed (mostly) by end-user devices, consequently, 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, the latency in communication from the clients to the DC as well as the security aspects that emerge from offloading data storage and computation to DCs operated by third parties 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 [**10.1145/242857.242867**], by taking into consideration computing and networking resources which exist beyond the boundaries of DCs and close to the edge of systems [**Leitao2018**] [**7488250**]. This paradigm aims at enabling the creation of systems which could otherwise be unfeasible with Cloud Computing: Google's self-driving car generates 1 Gigabyte every second [**datafloq**], and a Boeing 787 produces data at a rate close to 5 gigabytes per second [**finnegan_2013**], 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 characteristics of computational devices ranging from Edge Data Centers to 5G towers and mobile devices. These devices, contrary to the cloud, have heterogenous computational capacity and potentially limited and unreliable data lines. Given this, developing an efficient resource management and sharing platform which enables the adequate and efficient use of these devices is an open challenge for fully realizing Edge Computing.

1.2 Context

Resource management and sharing platforms are extensibly used in Cloud systems (e.g. Mesos [**hindman2011mesos**], Yarn [**Vavilapalli2013ApacheHY**], Omega [**41684**], 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) performing resource allocations to satisfy the needs of such tenants; (4) adapting to dynamic workloads such that the system remains balanced and system policies as well as performance criteria 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 which hinders the scalability of such such solutions, making them unsuitable for the scale of Edge Computing systems. Instead, we argue in favor of decentralized architectures for such resource management and sharing platforms, composed of multiple components, potentially organized in a flexible hierarchical way, and promoting load management decisions supported by partial and localized knowledge of the system.

Given this, it is paramount that devices cooperatively materialize a robust lightweight decentralized resource control system, which tracks resource demands and allocations. In such a system, the accuracy and freshness of the information each component has dictates how efficiently they manage resources such that the system remains balanced, and applications running on the infrastructure maintain their target quality of service. This system must federate devices such that they leverage on heterogeneity to build a hierarchical infrastructure which combines naturally with the device taxonomy, and adapts to the environment changes.

1.3 Contributions

1.4 Document structure

This remaining of this document is structured as follows:

Chapter ?? 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.

Chapter ?? further elaborates on the proposed contributions and the proposed work plan for the remainder of the thesis, including a more detailed explanation of our plans to experimentally evaluate our solution.

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 **??**:

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 **??**). Next, we study execution environments for applications, namely virtual machines and

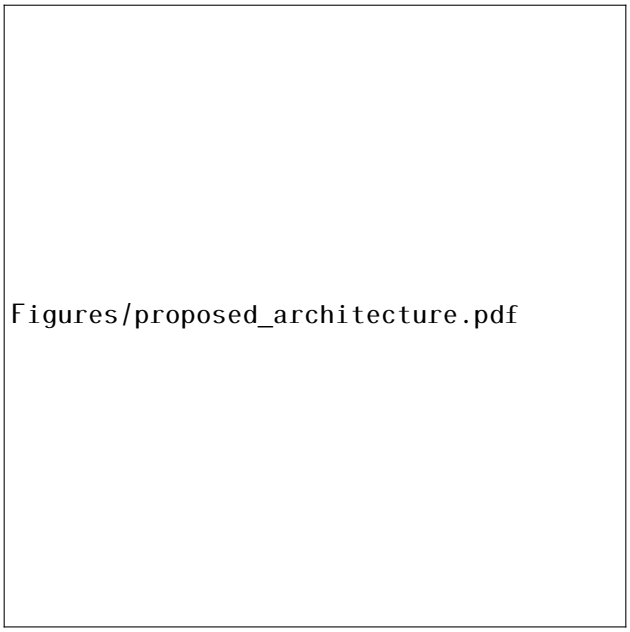

Figure 2.1: High-level architecture for a resource sharing platform

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

Following, we study how to federate devices in an efficient abstraction layer that establishes an efficient topology (Section **??**), 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 **??**). 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 **??**). 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 [**Leitao2018**]. It has the potential of enabling novel edge-enabled applications along with optimizing existing systems [**7488250**], making them more responsive.

Many approaches have already leveraged on some form of Edge computing in the past. **Cloudlets** [**10.1145/2307849.2307858**] 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 [**peng2004cdn**] 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 [**bonomi2012fog**] 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 [**villari2016osmotic**] 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 [**mobile_edge_cloud**] (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 **Leitao2018**, 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

Level	Category	Availability	Capacity
L4	Priv. Servers & Desktops	Medium	Medium
L5	Laptops	Low	Medium
L6	Mobile devices	Low	Low
L7	Actuators & Sensors	Varied	Low

Table **??** 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 [**1430629**].

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 [**docker**], Linux Containers [**lxc**], 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 [**7036275**].

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 [**preeth2015evaluation**], and contrary to VMS, restarting a container doesn't require rebooting the OS [**7036275**], meaning that a small-sized computation task may be accomplished much faster.

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

2.2.3 Discussion

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

2.3 Topology Management

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

In a P2P system, peers contribute to the system with a portion of their resources, so that the overall system can accomplish tasks which would otherwise be impossible for

a single peer to solve. Typically, this is achieved in a decentralized way, which means peers must establish neighboring connections among themselves to enable information exchange which, in turn, enables to progress towards the system goals.

Participants in a P2P system may know all other peers in the system, which is typically referred to as **full membership** knowledge, this is a popular approach in Cloud systems. However, as the system scales to larger numbers of peers, concurrently entering and leaving the system (a phenomenon called churn [**stutzbach2006understanding**]), 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 [**leitaoPHDthesis**]. 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 **??**, 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 **??** we ilustrate three examples of unstructured overlay networks: (A) is a representation of an overlay network where the connections are unidirectional (e.g. Cyclon [**jelasity2007gossip**]), 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.

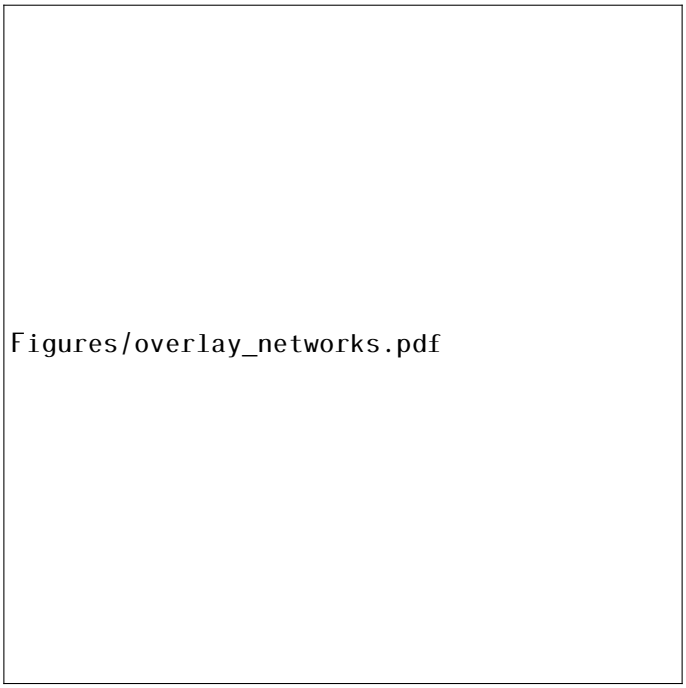

Figure 2.2: Examples Overlay Networks

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 [**Hyparview**] 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 [**leitao2014overnesia**]). 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 **??** illustrate three kinds of structured overlay networks: (D) corresponds to a tree, trees are widely used to perform broadcasts (e.g., PlumTree [**leitao2007epidemic**]) 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 [**Renesse2003**])). However, trees are very fragile in the presence of faults [**leitao2007epidemic**].

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 [**stoica2003chord**]), however, not all distributed hash tables rely on rings to perform effective routing. For example, in Kademlia [**maymounkov2002kademlia**], 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 [**rollerchain**]). 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 [**leitaoPHDthesis**]:

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 [**jelasity2005t**] 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 [**liang2005mon**] (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 [**Hyparview**] (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 [**leitao2007epidemic**]). 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 [**leitao2012x**] 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 [**Hyparview**]. 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 [**leitao2014overnesia**] 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 [**stoica2003chord**] is a well known structured overlay network where the protocol builds and manages a ring topology, similar to overlay (E) in Figure **??**. 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 [**dht_performance_churn**], 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 [**rowstron2001pastry**] 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 [**10.1007/3-540-45546-9_3**] and file storage systems like PAST [**990064**]. 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 [**tapestry**] Is a DHT similar to Pastry [**rowstron2001pastry**], 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 [**maymounkov2002kademlia**] 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 [**rowstron2001pastry**]) 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 [**gupta2003kelips**] 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 [**rollerchain**] 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 [**leitaoPHDthesis**], 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 [**leitaoPHDthesis**]: **(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 [**cohen2003incentives**].

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** [**leitaoPHDthesis**].

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 [**1022239**], this technique called a **random guided walk**. A common approach to bias random walks is to use bloom filters [**5751342**], 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 [**gnutella_gtk**]) 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 [**Chawathe2003**], employed towards effectively reducing the number of peers that have to disseminate queries on the second version of Gnutella [**gnutella_gtk**].

SOSP-Net [**garbacki2007optimizing**] (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**) [**leitaoPHDthesis**].

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 [**lakshman2010cassandra**] and Amazon's Dynamo [**decandia2007dynamo**] 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.

leitao2008large proposes a decentralized device monitoring system by employing Hyparview [**Hyparview**] 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 [**docker**] has a built tool called **Docker Stats** [**docker_stats**] 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 [**cAdvisor**] (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) [**cloudviz_2019**] 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 [**8613952**] [**DBLP:journals/corr/abs-1110-0725**]. 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*.

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

Table 2.2: Decomposability and duplicate sensitiveness of aggregation functions

Decomposability

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

Intuitively, a decomposable aggregation function is one where a function may be defined as a composition of other functions. Decomposable functions may **self-decomposable**, where the aggregated value is the same for all possible combinations of all sub-multisets partitioned in the multiset. This happens whenever the applied function is commutative and associative (e.g. min, max, sum, count). A canonical example of a decomposable function that is not self-decomposable is average, which consists in the sum of all pairs divided by the count of peers that contributed to the aggregation.

Duplicate sensitiveness

The second property of aggregation is **duplicate sensitiveness**, and it is related to wether a given value occurs several times in a multiset. Depending on the aggregation function used, the presence of repeated values may influence the result, it is said that a function is **duplicate sensitive** if the result of the aggregation function is influenced by the repeated values (e.g. SUM). Conversely, if the aggregation function is **duplicate insensitive**, it can be successfully repeated any number of times to the same multiset without affecting the result (e.g. MIN and MAX). Table **??** classifies popular aggregation functions in function of decomposability and duplicate sensitiveness as found in [**DBLP:journals/corr/abs-1110-0725**].

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 [**8613952**]. 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 [**gossip_aggregation**], 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 [**Renesse2003**] 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 [**massie2004ganglia**] 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 [**SDIMS**] (Scalable Distributed Information Management System) proposes a combination of techniques employed in Astrolabe [**Renesse2003**] and distributed hash tables (in this case, Pastry [**rowstron2001pastry**]). 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 [**prometheus**] 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 [**SDIMS**] 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 [**Renesse2003**] 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 .

citation

citation

citation

citation

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.

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 **??** 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.

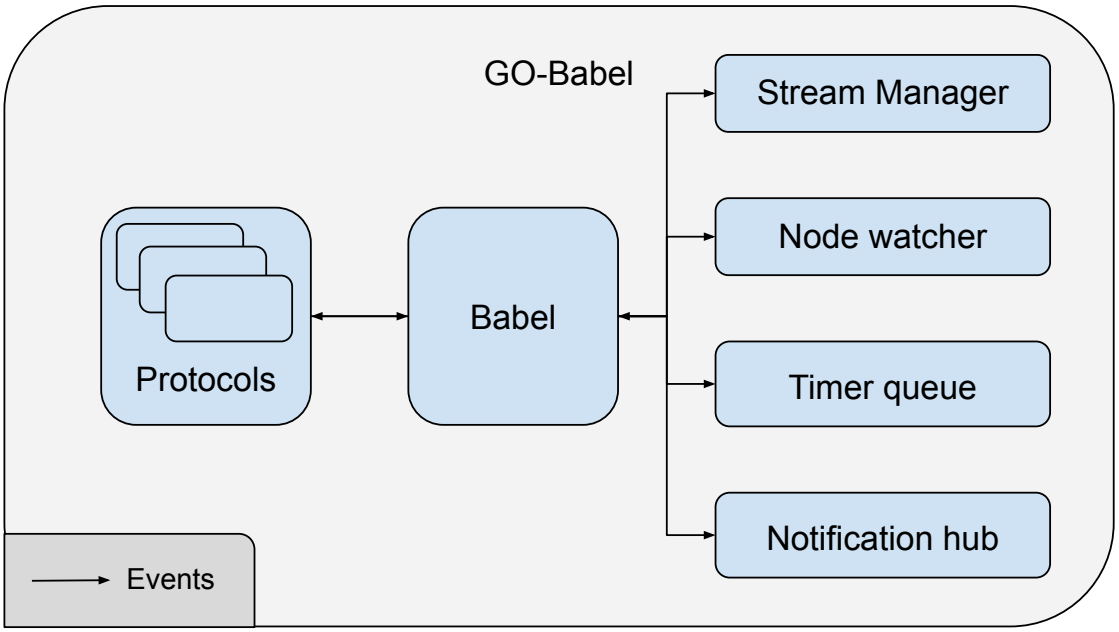

Figure 3.1: An overview of the architecture of GO-Babel

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 Monitoring Overlay Network) is an overlay network aiming to create logical connections among nodes integrating the network, forming multiple tree-shaped networks. Then, it provides an API to request information about nodes and services running in the system, which is collected on-demand by the monitoring protocol via efficient information aggregation and dissemination using the tree structure.

In this chapter, we will begin by explaining the targeted environment and the operation of the overlay network, whose tree shape is the basis for the aggregation protocol. After, detail how the aggregation protocol performs aggregations in the tree, and lastly, list the operations exposed by the API and discuss how it interacts with the remaining components.

insert refs to subsections ahead

This solution, as observable in figure **??**, is composed of three major components:

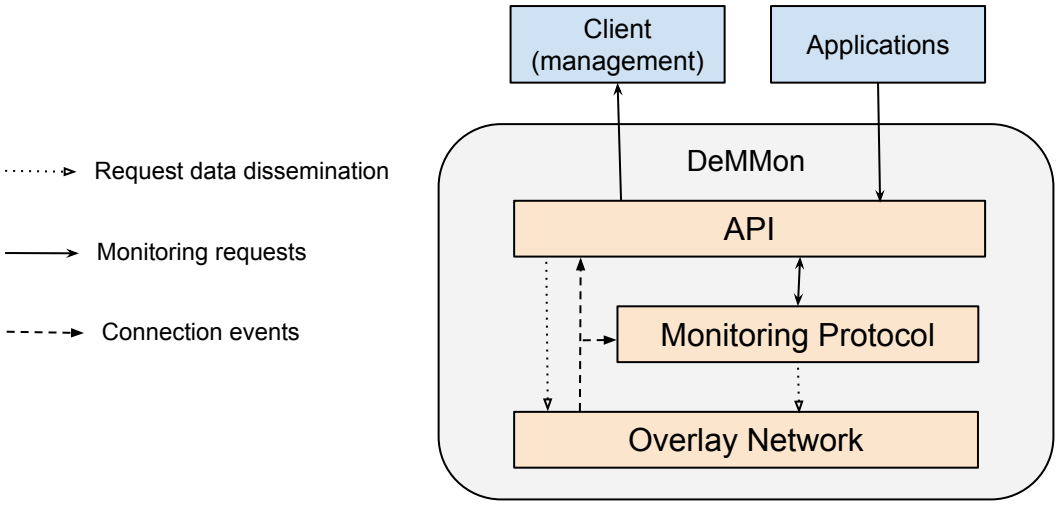

Figure 4.1: An overview of the architecture of DeMMon

1. The overlay network, which strives to build a multi-tree-shaped network, nodes in

this network use latency, node capacity, and a set of logical rules to change their location either from one tree to another or within their own tree.

2. The monitoring protocol, which is a component that collects and disseminates information using the overlay network's established connections. It communicates via notifications and asynchronous request-replies with the overlay network to receive updates regarding established connections and connection failures. Lastly, it receives requests from the API to collect information.

3. Lastly, the API receives updates from both the overlay network and the monitoring protocol, exposes the received information from those layers, and allows ingestion of new information. Furthermore, it allows issuing commands to collect new information, perform local aggregations periodically, or trigger issued alarms based on the respective conditions.

4.1 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.

4.1.1 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.

4.1.2 Overview

As previously mentioned, the main objective of the created protocol is to establish a latency and capacity-aware multi-tree-shaped network, rooted on the previosuly 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

put paragraph of the intended resulting structure

The devised membership protocol is coalesced by 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.

4.1.2.1 Join mechanism

The joining mechanism is the mechanism responsible for choosing the initial parent connection, which performs a greedy depth-first search to find a suitable low latency parent. This mechanism is the first to be executed by all nodes in the system, with the pseudocode presented in **??**.

Its first step (line **??**) 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, (4) the best node contacted so far in the join process, (5) 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 latency measured, its current parent, number of children, whether the node replied to the message, its IP, and an array of its childrens' IP and children number.

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

Algorithm 1 Join Protocol

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

message. Landmarks that receive this message will send a reply and establish a connection back (line **??**), a 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 **??**), then the node awaits the responses from the contacted nodes, during this process, the joining node listens for any "joinTimers" which have triggered, or until any of the node measurements has been unsuccessful (meaning contacted nodes have exceeded their reply timeout), if this happens, in the case of the contacted node being a landmark, the joining node aborts the join process and waits a configurable amount of time until attempting to re-join the overlay again. If the timed-out node is not a landmark, then that node is excluded from the remaining of the join process, and the join process is resumed as normal (line **??**).

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 **??**). When the joining node receives the joinReply, it checks to see if it is not from a timed out node, or if the node's parent is not the same anymore, if any of these conditions are observed, then the reply is discarded.

Then, whenever the joining node has either: received the JOINREPLY messages from all the contacted nodes, and stored the information (line **??**), or they have been timed-out via the "joinTimers", it evaluates all contacted nodes, attempting to find the contacted node with lowest latency by performing the following verifications:

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

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

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 (line **??**).

If none of the verified peers were suitable to start a new recursive step (line **??**) (either had no children or 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. If there is no previous node, the joining node aborts the process and restarts the join process later.

The join process finishes with the reception of a "JoinRequestReply" and the establishment of a connection between the sender and receiver of the message. If this does not

happen, and JoinRequestTimer triggers, then the node will recursively fall back to the parent (if it exists) or re-join the overlay later.

4.1.2.2 Active view maintenance

4.1.2.3 Passive view maintenance

Algorithm 3 Membership protocol (Oportunistic Optimization)

```
1:  State
2:      eView : set<Node>
3:
4:  Every config.RandWalkPeriodicity Do
5:      ascNeighs, allNeighs = set
6:      ascNeighs = ascNeighs + parent + siblings
7:      allNeighs = allNeighs + ascNeighs + children
8:      sample = getRandSample(eView + allNeighs, config.NrPeersToMergeRandWalk)
9:      sendMessage(RandomWalk<sample, config.RandWalkTTL, self.ID, self.IP>, getRand(ascNeighs))
10:
11: Every config.OportunisticOptimizationTimeout Do
12:     toMeasureRand = getRandSample(eView, config.NrPeersToMeasureRandom)
13:     toMeasureBiasedOpts = sortByEuclideanDist(eView / toMeasureRand)
14:     toMeasureBiased = getRandSample(toMeasureBiasedOpts, config.NrPeersToMeasureRandom)
15:     for p do in toMeasureRand:
16:         measurePeer(p)
17:     for p do in toMeasureBiased:
18:         measurePeer(p)
19:
20: Upon receive( RandomWalk<sample, ttl, nID, orig>, sender) Do
21:     stepsTaken = config.RandWalkTTL - ttl
22:     nrToAdd = config.NrPeersToMergeRandWalk
23:     nrToMerge = config.NrPeersToMergeRandWalk
24:     ascNeighs, allNeighs = set(), set()
25:     ascNeighs = ascNeighs + parent + siblings
26:     allNeighs = allNeighs + ascNeighs + children
27:     if stepsTaken < config.NrStepsToIgnore then:
28:         nrToMerge = 0
29:     toAdd = getRandSample(excludeDescendantsOf([eView + allNeighs] / sample),nID), nrToAdd)
30:     toRemoveFromSample = getRandSample(sample, nrToMerge)
31:     sample = sample / toRemoveFromSample
32:     sample = sample + toAdd
33:     target = getRand(excludeDescendantsOf(allNeighs, nID)
34:     if target == nil || ttl == 0 then
35:         sendMessageSideChannel(RandomWalkReply<sample>, orig)
36:     else
37:         sendMessage(RandomWalk<sample, ttl-1, nID, orig>, getRandom(ascNeighs))
38:     eView = excludeDescendantsOf(toRemoveFromSample, self.ID) + eView
39:     eView = eView / allNeighs
40:     eView = eView[:config.MaxEViewSize]
41:
42: Upon peerMeasured(p, latency) Do
43:     latencyImprovement := parent.measuredLatency - Latency
44:     if latencyImprovement >= config.MinLatencyForImprovement then
45:         sendMessageSideChannel(OportunisticImprovementReq<self>,p)
46:
47: Upon receive(OportunisticImprovementReq<p>,sender) Do
48:     if isDescendent(p.ID,self) or parent == nil then
49:         sendMessageSideChannel(OportunisticImprovementReqReply<false>,sender)
50:     else
51:         addChildren(sender)
52:         sendMessageSideChannel(OportunisticImprovementReqReply<true>,sender)
53:
54: Upon receive(OportunisticImprovementReqReply<answer>,sender) Do
55:     if answer then
56:         addParent(sender)
57:
58: Procedure isDescendentOf(nodeID, PotentialDescID)
59:     return PotentialDescID.Contains(nodeID)
60:
```

Algorithm 2 Membership protocol (Active view Optimization)

```
1:  State
2:      parent                                                              ▷ defined in join
3:      children
4:      childrenLatencies : dict<string:dict<string:number»
5:
6:  Every config.updatePeriodicity Do
7:      if parent != nil then
8:          sLatencies ⟵ set()
9:          for sibling in siblings do
10:             sLatencies.append(<sibling.IP,sibling.measuredLatency)
11:         sendMessage(UpdateChildStatus<children, siblingLatencies>, parent)
12:     for child in chidren do
13:         sendMessage(UpdateParentStatus<self,child.ID, parent>)
14:
15: Upon receive(UpdateParentStatus<parent,myID, grandParent>, sender) Do
16:     if sender == parent.IP then
17:         parent ⟵ parent
18:         self.ID ⟵ parent.ID + myID
19:         grandParent ⟵ grandParent
20:
21: Upon receive(UpdateChildStatus<child, childSiblingLatencies>, sender) Do
22:     if children[sender] != nil then
23:         childrenLatencies[sender]=childSiblingLatencies
24:
25: Every config.updateChildPeriodicity Do
26:     childrenLatValues ⟵ set()
27:     for c1 in children do
28:         for <c2, lat> in childrenLatencies[c] do
29:             if lat - c1.measuredLatency >= d.config.maxLatDowngrade then e break
30:             higherBwC ⟵ c1
31:             lowerBWC ⟵ c2
32:             if c2.bw > c1.bw then
33:                 higherBwC ⟵ c2
34:                 lowerBWC ⟵ c1
35:             childrenLatValues.add(<higherBwC,lowerBWC,lat>)
36:     kickedNodes, newParents ⟵ set(),set()
37:     potentialChildren ⟵ dict<string,set<Node>>
38:     sortByLatency(childrenLatValues)
39:     for <higherBwC,lowerBWC,lat> in childrenLatValues do
40:         if len(children) - len(kickedNodes) <= config.MinGroupSize then
41:             break
42:         if higherBwC in kickedNodes or lowerBWC in kickedNodes then
43:             continue
44:         if loserBWC in newParents then
45:             continue
46:         if higherBwNode.nrChildren == 0 then
47:             potentialChildren[higherBwNode].append(lowerBWC)
48:             if len(potentialChildren) >= config.MinGroupSize then
49:                 for potentialChild in potentialChildren[higherBwNode] do
50:                     newParents <- newParents + higherBwNode
51:                     send(OptimizationPropose<higherBwNode>, potentialChild)
52:                     higherBwNode.nrChildren++
53:                     kickedNodes <- kickedNodes + potentialChild
54:                 for <nIP,pontentialChildrenTmp> in potentialChildren do
55:                     pontentialChildrenTmp.deleteAll(potentialChildren[higherBwNode])
56:                 potentialChildren[higherBwNode] ⟵ set<Node>
57:                 continue
58:         kickedNodes <- kickedNodes + lowerBWC
59:         send(OptimizationPropose<higherBwNode>, lowerBWNode)
60:
61: Upon receive(OptimizationPropose<newParent>, sender) Do
62:     if sender == parent then
63:         send(OptimizationProposeRequest<sender>, newParent)
64:
65: Upon receive(OptimizationProposeRequest<p>, sender) Do
66:     if  p == parent && sender in siblings then          ▷ parent issuing the message is the same parent that i have
67:         send(OptimizationProposeRequestReply<true>, sender)
68:     else
69:         sendSideChannel(OptimizationProposeRequestReply<false>, sender)
70:
71: Upon receive(OptimizationProposeRequestReply<reply>, sender) Do
72:     if reply then
73:         sendMessageAndDisconnectFrom(DisconnectMessage<>, parent)
74:         addParent(sender)
75:
```

4.1.3 Summary

4.2 Monitoring protocol

4.2.1 Overview

4.2.2 Aggregation mechanisms

4.2.2.1 Single root aggregation

4.2.2.2 Multi root aggregation

4.2.2.3 Neighborhood aggregation

4.2.3 Summary

4.3 API

4.3.1 System Model

4.3.2 Overview

4.3.3 Showcase

5

Benchmark

6

Evaluation

7

Conlusions and future work