Nuno Miguel Cadima Vasconcelos Morais

Bachelor in Computer Science and Engineering

DeMMon: Decentralized Edge Management and Monitoring

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

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Fevereiro, 2020

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

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. Visto 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

1 Introduction **1**

1.1 Motivation 1

1.2 Context 2

1.3 Expected Contributions 3

1.4 Document structure 3

2 Related Work **5**

2.1 Edge Environment 6

2.1.1 Edge Computing 6

2.1.2 Edge Environment Taxonomy 7

2.1.3 Discussion 8

2.2 Execution Environments 9

2.2.1 Virtual Machines 9

2.2.2 Containers 9

2.2.3 Discussion 10

2.3 Topology Management 10

2.3.1 Taxonomy of Overlay Networks 11

2.3.2 Overlay Network Metrics 13

2.3.3 Examples of Overlay Networks 13

2.3.4 Discussion 17

2.4 Resource Location and Discovery 17

2.4.1 Querying techniques 18

2.4.2 Centralized Resource Location 18

2.4.3 Resource Location on Unstructured Overlays 18

2.4.4 Resource Location on Distributed Hash Tables 19

2.4.5 Discussion 20

2.5 Resource Monitoring 20

2.5.1 Device Monitoring 21

2.5.2 Container Monitoring 21

2.5.3 Aggregation 22

2.5.4 Aggregation techniques 23

2.5.5 Monitoring systems 23

2.5.6 Discussion 25
2.6 Resource Management 26
2.6.1 Resource Management Taxonomy 26
2.6.2 Resource Sharing Systems 27
2.6.3 Discussion 30
2.7 Summary 30

3 Planning **31**
3.1 System Model 31
3.2 Proposed Solution 32
3.3 Evaluation 33
3.4 Scheduling 34

Bibliography **37**

Chapter 1

Introduction

1.1 Motivation

Nowadays, the Cloud Computing paradigm is the standard for development, deployment and management of services, and most of the software systems present in our everyday life, such as Google Apps, Amazon, Twitter, among many others, is deployed on some form of cloud service. Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and software systems in the data centers that provide those services [1]. It provides the illusion of unlimited computing power, which revolutionized the way developers, companies, and users rationalize about applications.

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 [39]. 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 [21], by taking into consideration computing and networking resources which exist beyond the boundaries of DCs and close to the edge of systems [37] [54]. 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 [53], and a Boeing 787 produces data at a rate close to 5 gigabytes per second [18], 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 [24], Yarn [60], Omega [52], 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 Expected Contributions

The expected contributions from the work to be conducted in this thesis, as will be further detailed in Chapter 3, arise from the aforementioned challenges. We plan to focus on creating a decentralized resource monitoring and management solution tailored for provisioning resources for edge-enabled applications. Given this, the expected contributions to arise from our work consist of:

- A novel decentralized overlay network protocol tailored towards federating large numbers of edge devices in a hierarchical way.
- A lightweight monitoring solution which relies on that overlay structure to aggregate information regarding applications' operation and the load across edge resources.
- A decentralized resource management solution capable of satisfying resource allocations for multi-tenant resource sharing, based on the monitoring information captured by our platform
- An extensive experimental assessment of our solution that will demonstrate its correctness and compare its performance with regard to existing solutions in a realistic test bed. Parts of this experimental work will be based on simple mock ups of applications that will be implemented to exercise our platform.

1.4 Document structure

This remaining of this document is structured as follows:

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

Chapter 3 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.

Chapter 2

Related Work

The goal of this chapter is to present the related work studied that is associated with our objectives. We begin by identifying the four high-level requirements of a resource sharing platform, as denoted in figure 2.1:

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

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

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

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

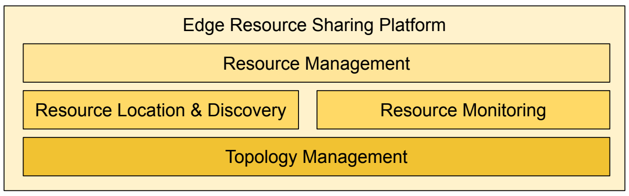

Figure 2.1: High-level architecture for a resource sharing platform

Considering the identified high-level components of such a system, in the following sections we cover the taxonomy of devices which compose the edge environment, and discuss how they can be employed towards the design of the proposed solution (Section 2.1). Next, we study execution environments for applications, namely virtual machines and containers, discuss their performance impact as well as their strengths and limitations towards supporting edge-enabled applications (Section 2.2).

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

Next, we cover tools to collect metrics from the aforementioned execution environments that are relevant towards performing efficient resource allocations. We analyze how to aggregate those metrics in a decentralized manner, and discuss relevant resource monitoring systems in the literature, for each, we address its limitations and advantages for the edge environment (Section 2.5). Lastly, we cover the taxonomy of resource management solutions, and present popular systems in the literature that share aspects with the solution we aim at developing (Section 2.6).

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 [37]. It has the potential of enabling novel edge-enabled applications along with optimizing existing systems [54], making them more responsive.

Many approaches have already leveraged on some form of Edge computing in the past. **Cloudlets** [61] 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 [46] are specialized high bandwidth servers strategically located at the edge of the network which replicate content from a certain origin and serve it at reduced latencies, effectively decentralizing the content delivery.

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

Osmotic Computing [63] 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 [42] (MEC) is a network architecture which proposes to provide fast-interactive responses for mobile applications. It solves this by employing the devices in the edge (e.g. base stations and access points) to provide compute resources for latency-critical mobile applications (e.g. facial recognition). Similar to Osmotic Computing, we consider MEC a subset of edge computing, given that its primary focus is on how to offload the computation from mobile to the cloud and not vice-versa.

2.1.2 Edge Environment Taxonomy

According to Leitão et al. [37], edge devices may be classified according to three main attributes: **capacity** refers to computational, storage and connectivity capabilities of the device, **availability** consists in the probability of a device being reachable, and finally, **domain** characterizes the way in which a device may be employed towards applications, either by performing actions on behalf of users (user domain) or performing actions on behalf of applications (applicational domain). Given that the concern of our work is towards building the underlying infrastructure for these applications, we will only focus on capacity and availability when classifying the taxonomy of the environment.

Table 2.1: Taxonomy of the edge environment

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

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

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

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

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

Levels 6 consists of *tablets* and *mobile devices*, which have low capacity, availability, and short battery life. Given this, they are limited in how they can perform contribute towards edge applications. Aside from caching user data, they may filter or aggregate of data generated from devices in level 7. Finally, **level 7** consists of *actuators*, *sensors* and *things*, these devices are the most limited in their capacity, and enable limited forms of computation in the form of aggregation and filtering.

2.1.3 Discussion

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

Devices in levels 0-5 are potential candidates towards building the resource management and monitoring system we intend to create. The low availability and potential mobility of devices in higher levels make them unsuitable, as they could potentially be a source of instability in the system. This effect can be circumvented by employing devices in other levels as gateways for those devices, hence starting to establish a hierarchy on the way different application components interact.

2.2 Execution Environments

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

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

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

2.2.1 Virtual Machines

A VM provides a complete environment in which an operating system and many processes, possibly belonging to multiple users, can coexist. By using VMs, a single-host hardware platform can support multiple, isolated guest operating system environments simultaneously [55].

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 [17], Linux Containers [27], among others) can be considered as a lightweight alternative to hypervisor-based virtualization. When using containers, applications share an OS (and maybe binaries and libraries), and implement isolation of processes at the operating system level. As a result, these deployments are significantly smaller in size than hypervisor deployments, for comparison, a physical machine may store hundreds of containers versus a few tens of VMs [2].

In terms of performance, container-based virtualization can be compared to an OS running on bare-metal in terms of memory, CPU, and disk usage [47], and contrary to VMS, restarting a container doesn't require rebooting the OS [2], meaning that a small-sized computation task may be accomplished much faster.

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

2.2.3 Discussion

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

2.3 Topology Management

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

In a P2P system, peers contribute to the system with a portion of their resources, so that the overall system can accomplish tasks which would otherwise be impossible for a single peer to solve. Typically, this is achieved in a decentralized way, which means peers must establish neighboring connections among themselves to enable information exchange which, in turn, enables to progress towards the system goals.

Participants in a P2P system may know all other peers in the system, which is typically referred to as **full membership** knowledge, this is a popular approach in Cloud systems. However, as the system scales to larger numbers of peers, concurrently entering and leaving the system (a phenomenon called churn [57]), this information becomes costly to maintain up-to-date.

In order to circumvent the aforementioned problems, a common alternative is to have peers only maintain a view of a subset of all peers in the system, which is called **partial membership**. This information is maintained by some membership algorithm which restricts neighboring relations among peers. Partial membership solutions are attractive

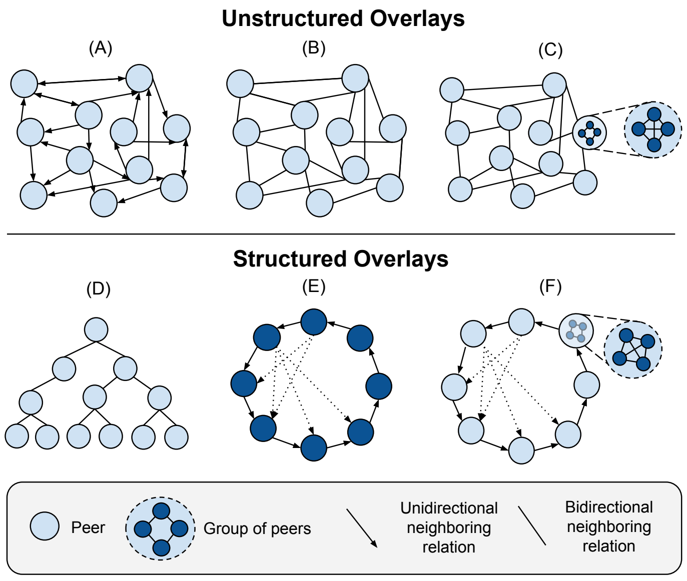

Figure 2.2: Examples Overlay Networks

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

2.3.1 Taxonomy of Overlay Networks

Overlay networks are logical networks which operate at the applicational level, these rely on an existing network (commonly referred to as the *underlay*) to establish neighboring relations, where each participant tipically only communicates directly with its overlay neighbors [59]. Overlays are commonly designed towards specific applicational needs, as such, their neighboring relations may or may not follow some sort of logic. As observable in Figure 2.2, there are two main categories of overlays: **structured** and **unstructured**:

Unstructured Overlays

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

In figure 2.2 we ilustrate three examples of unstructured overlay networks: (A) is a representation of an overlay network where the connections are unidirectional (e.g. Cyclon [29]), in this type of overlay peers have no control over the status of incoming connections, consequently, a peer may become isolated from the network without realizing it, which is undesirable.

Overlay (B) is similar to (A), however, neighboring connections are bidirectional. This means that a peer with a given number of outgoing connections must also have the correspondent number of incoming connections, diminishing the risk of the peer becoming disconnected from the overlay (this is the approach taken by HyParView [33] 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 [38]). Grouping multiple devices into a group can be useful because: (1) failures can be quickly identified and resolved by other members of the group; (2) nodes can replicate data within the group, leading to increased availability of that data; (3) for devices with low computing capabilities, groups are useful because nodes have nearby neighbors which can simplify the offload of computational tasks.

Structured Overlays

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

In Figure 2.2 illustrate three kinds of structured overlay networks: (D) corresponds to a tree, trees are widely used to perform broadcasts (e.g., PlumTree [34]) 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 [49])). However, trees are very fragile in the presence of faults [34].

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 [56]), however, not all distributed hash tables rely on rings to perform effective routing. For example, in Kademlia [44], 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 [45]). 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 [59]:

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 [28] 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 [41] (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 [33] (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 [34]). 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 [36] 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 [33]. 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 [38] 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 [56] is a well known structured overlay network where the protocol builds and manages a ring topology, similar to overlay (E) in Figure 2.2. Each node is assigned an m-bit identifier that is uniformly distributed in the id space. Then, peers are ordered by identifier in a clockwise ring, where any data piece identified by k, is assigned to the first peer whose identifier is equal or follows k in the identifier space.

Chord implements a system of "shortcuts" called the *finger table*. The finger table contains at most m entries, each ith entry of this table corresponds to the first peer that succeeds a certain peer n by 2^{ith} in the ring. This means that whenever the finger table is up-to-date, and the system is stable, lookups for any data piece only take logarithmic time to finish.

Although Chord provides the a good trade-off between bandwidth and lookup latency [40], 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 [50] 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 [51] and file storage systems like PAST [16]. 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 [66] Is a DHT similar to Pastry [50], 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 [44] 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 [50]) 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 [23] 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 [45] 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 [59], 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 [59]: **(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 [10].

Although centralized architectures are widely used nowadays, they lack the necessary scalability to index the large number of dynamic resources we intend to manage, and have limited fault tolerance to failures, 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** [59].

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 [12], this technique called a **random guided walk**. A common approach to bias random walks is to use bloom filters [58], 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 [22]) 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 [8], employed towards effectively reducing the number of peers that have to disseminate queries on the second version of Gnutella [22].

SOSP-Net [19] (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) [59].

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 [32] and Amazon's Dynamo [14] are widely used implementations of one-hop DHTs.

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

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

2.4.5 Discussion

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

Unstructured resource location systems are attractive to perform queries that search for resources which are abundant in the system, however, this approach is inefficient when performing exact match queries, as a finding the exact resource in an unstructured resource location system requires flooding the entire system with messages. Conversely, distributed hash tables are especially tailored towards exact match queries, but are less robust to churn and are subject to low-capacity nodes being a bottleneck in routing procedures.

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

2.5 Resource Monitoring

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

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

2.5.1 Device Monitoring

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

Leitao et al. [35] proposes a decentralized device monitoring system by employing Hyparview [33] 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 [17] has a built tool called **Docker Stats** [15] 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 [20] (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) [9] 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 [11] [31]. It can be employed, for example, towards computing the average of available computing resources in a certain part of the network, or towards identifying application hotspots by aggregating the average resource usage in certain areas, among many other uses. There are two properties of aggregation functions: *decomposability* and *duplicate sensitiveness*.

Decomposability

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

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

Duplicate sensitiveness

The second property of aggregation is **duplicate sensitiveness**, and it is related to wether a given value occurs several times in a multiset. Depending on the aggregation function used, the presence of repeated values may influence the result, it is said that a function is **duplicate sensitive** if the result of the aggregation function is influenced by the repeated values (e.g. SUM). Conversely, if the aggregation function is **duplicate insensitive**, it can be successfully repeated any number of times to the same multiset without affecting the result (e.g. MIN and MAX). Table 2.2 classifies popular aggregation functions in function of decomposability and duplicate sensitiveness as found in [31].

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

Table 2.2: Decomposability and duplicate sensitiveness of aggregation functions

2.5.4 Aggregation techniques

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

Hierarchical aggregation

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

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

Hierarchical approaches, due to taking advantage of device heterogeneity, are attractive in edge environments. However, due to the low computational power of devices, not all nodes may be able to handle the additional overhead of maintaining the hierarchical topology.

Continuous aggregation

Continuous aggregation consists in the continuous computation and exchange of partial averages data among all active nodes in the aggregation process [11]. 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 [30], 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 [49] 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 [43] 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 [65] (Scalable Distributed Information Management System) proposes a combination of techniques employed in Astrolabe [49] and distributed hash tables (in this case, Pastry [50]). 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 [48] 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 [65] 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 [49] 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 Resource Management

In this section we study resource management, which consists in providing resources (e.g. computing power, memory, among other) to tenants (i.e. applications, frameworks, among others) such that tenants can perform computations. In this section we cover the different types of resource management solutions and study popular solutions in the literature.

2.6.1 Resource Management Taxonomy

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

2.6.1.1 Tenancy

The term tenancy in resource management refers to whether or not underlying hardware resources are shared among entities [25].

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

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

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

2.6.1.2 Control

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

Centralized control consists in a centralized component with a global view of the state of the system making all decisions regarding resource allocations. Intuitively, given that a centralized component generates manages all the resources in the system, this component can easily enforce policies to achieve the desired performance guarantees or fairness goals by identifying and only throttling the tenants or system activities responsible for resource bottlenecks [62].

Decentralized control architectures are defined by having the decision-making process regarding resource allocations distributed across multiple components [25]. This topic has yet not been subject to much research, although it is of extreme relevance towards edge environments. For example, if the system is globally distributed, it may take too long for a centralized controller to identify hotspots in a certain zone and load-balance them.

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

2.6.2 Resource Sharing Systems

Mesos [24] is a multi-tenant centralized resource sharing platform which attempts to provide fine-grained resource sharing in a data center. The tenants for this platform are frameworks such as HDFS [4], MapReduce [13], among others, which in turn support multiple applications running within a DC. In short, the Mesos resource sharing system consists of a *master* process which manages *slave* daemons running on each cluster node. In order to achieve fault-tolerance for the master component, Mesos employs Zookeeper [26] to maintain replicas, elect a new master, and transfer state to a new master in case the active master fails.

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

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

Yarn (Yet Another Resource Negotiator) [60] is a centralized multi-tenant resource sharing platform which attempts decouple the programming model from the resource management infrastructure, and delegate many scheduling functions to per-application components. The architecture of YARN is composed by: a per-cluster Resource Manager (RM), multiple Application Masters (AM), and Node Managers (NM). The Resource Manager (RM) tracks resource usage and node liveness, enforces allocation invariants and arbitrates contention among tenants.

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

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

Omega [52] is a scheduler designed for grid computing systems composed by schedulers and workers. Each scheduler receives large amounts of jobs composed by either one or many tasks that have to be scheduled among workers. Contrary to YARN, which is monolithic, OMEGA uses multiple schedulers per cluster, each with a shared global view of the cluster state.

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

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

Edge NOde Resource Management [64] (ENORM) is framework aimed at employing edge resources towards applications by provisioning and auto-scaling edge node resources. ENORM proposes a three-tier architecture: (1) the Cloud tier, where application servers are hosted; (2) the middle tier, where the edge nodes are situated; (3) the bottom tier, where user devices (e.g. smartphones, wearables, gadgets) are situated.

To enable the use edge nodes, ENORM deploys a cloud server manager on each application server, which communicates with potential edge nodes requesting computing services, deploys partitioned servers on the edge nodes, and finally receives updates from the edge nodes to update the global view of the application server on the cloud.

ENORM authors tested the designed system using an online game based on Pokemon GO (iPokemon)[7], the framework partitions the game server and sends user data relevant to the geographical location of the edge node. Users from the relevant geographical zone connect to the edge server and are serviced as if they were connected to the data center. Limitations from this framework are the large number of required information to perform deployments, and the lack of fault-tolerance and scalability, which may arise from employing a centralized component to perform monitoring and management of resources.

FogTorch [5] is a service deployment framework aimed at determining eligible deployments for an application over a given Fog infrastructure, modeled by: (1) Cloud Data Centers, which are denoted by their location and software capabilities they provided; (2) Fog Nodes, which consist of tuples containing the location, hardware and software capabilities and the things directly reachable from the fog node; (3) Things, which are represented by a tuple denoting the thing (sensor or actuator) location and its type; (4) QoS profiles, which are sets of QoS profiles consists of a set of pairs composed by the latency and bandwidth of a communication link. (5) Applications, which are composed of independent sets of components, each with a set of requirements regarding QoS profiles, hardware and software capabilities, and things.

The authors model the notion of service deployments as restrictions over the aforementioned system model and employ a greedy heuristic which reduces the search space of devices constituting options for these service deployments. FogTorch originated **FogTorchPI** [6], this solution employs the same system model, but instead of employing a greedy algorithm, employs Monte Carlo simulations to returns a set of eligible deployments along with their QoS-assurance, heuristic rank and resource consumptions.

FogTorch provides a comprehensive system model which is able to model many different types of application requirements, however, a limitation from the proposed service deployment strategy is that it requires a large amount of information, and a global up-to-date view of the system, limiting system scalability.

2.6.3 Discussion

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

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

Due to their low capacity, devices in the edge of the network are very susceptible to workload changes, for example, a 5G tower which is hosting a service cannot handle a drastic increase in the number of users it is serving. In this scenario, we argue that in order to maintain pre-established performance criteria, devices must autonomously make resource management decisions such as scaling horizontally or vertically in order quickly meet the demands of users.

2.7 Summary

In this chapter we discussed relevant state of the art found in the literature towards devising a resource management and monitoring platform tailored for edge environments. We began by studiying the devices which materialize 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 effective resource location and discovery in these overlays. After this, we examined related work towards collecting metrics in a decentralized manner regarding not only the execution of devices, but also the aforementioned execution environments. Finally, we covered popular resource management systems which perform resource scheduling based on these metrics.

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.

Chapter 3

Planning

In this Chapter we begin by defining the system model and the intuition for the proposed solution, followed by defining and discussing a set of metrics to verify its success through experimental work. In the last section we present the work plan for the remaining time of the thesis.

As previously mentioned, the challenge we propose to address is the creation of a large-scale decentralized management and monitoring infrastructure tailored for heterogenous edge devices. This infrastructure may be used to track the state of applications (for load balancing), discover nearby devices to offload tasks, or find a set of devices to deploy a new application in a strategic location, enabling in the future the autonomic management of edge-enabled applications.

3.1 System Model

As defined in Section 2.1, the edge environment is composed by devices classified in levels ranging from 0-7. From this classification, we outline two major categories:

Stable devices consist of devices ranging from levels 0-5 in the taxonomy. We consider devices in these levels “stable” because they are usually connected across a wired medium (except in the case of laptops), and have enough computational capacity towards performing monitoring and management of tasks.

Unstable devices comprises devices in levels six and 7 of the taxonomy. In the case of mobile devices (level 6), we consider them unstable due to their low computational power and the fact that their physical location may change rapidly, which may lead to additional overhead in tracking their location. Given that devices in levels 6 and 7 are connected across a wireless medium, raising a large number of concerns that are outside the scope of this work, we intend to employ only stable devices in order to materialize

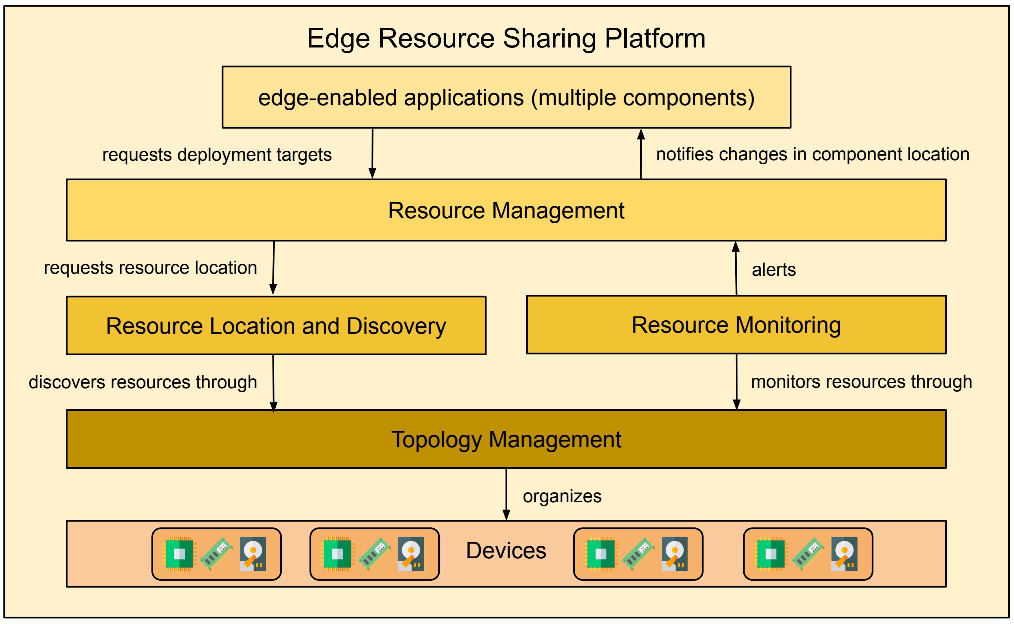
Edge Resource Sharing Platform
edge-enabled applications (multiple components)
requests deployment targets
notifies changes in component location
Resource Management
requests resource location
alerts
Resource Location and Discovery
Resource Monitoring
discovers resources through
monitors resources through
Topology Management
organizes
Devices

Figure 3.1: High-level architecture of the proposed resource sharing platform

our solution, unstable devices may employ stable devices as gateways to the system.

Following, we have applications. In the context of this thesis, we will focus on the management of **edge-enabled applications** running on containers. We consider these as applications which are decomposable into multiple independent components (all) be hosted in a single container and function as a *monolithic* application, or alternatively have components hosted by containers scattered throughout the system.

Consequently, we define a set of operations which are critical for this type of applications: *replicating*, *migrating* and *decomissioning*. Replicating consists of creating a copy of an application component and deploying it to another device in the system, migrating means transferring a component (or multiple components) of the application to another device, and decomissioning consists of removing / disabling an application component running in a given device. These three operations support the lifecycle of the applications, and enable both scaling horizontally through replication, and scaling vertically through migration (depending on the computational capabilities of the target device), in addition to allowing applications to improve their latency.

Finally, during the development of our solution, we assume that there is at most one simultaneous unexpected failure across the data centers belonging to the system, and assume that paths between any two nodes in the system have the same chance of failure regardless of the administrative domains they belong to.

3.2 Proposed Solution

In Figure 3.1, we illustrate the proposed architecture of the solution we intend to design, which is composed of four co-dependent mechanisms exercised by every participant of the system (that executes the management solution).

At the bottom, we have **topology management**, whose responsibilities consist of: (1) ensuring that devices belonging to the overlay remain connected at all times; (2) materializing a hierarchical topology based on the capabilities of devices composing the system; (3) assuring that devices have at least one non-faulty device connected to it; (4) detecting failures of the participating nodes. This mechanism, inspired in works such as Astrolabe [49], enables the correct behavior of the remaining components.

Following, we have **resource monitoring**, the objectives of this mechanism consist of: (1) collecting metrics about the containers hosting the applicational components; (2) aggregating the collected data in a decentralized manner; (3) deciding whether an application component hosted on certain device is not performing according to the established performance criteria; (4) alerting the resource management mechanism about failures in the containers and components performing sub-optimally.

The **resource management** mechanism handles the alerts emitted by the resource monitoring mechanism and, depending on the alert, immediately relocates application components to ensure they remain functional. As previously mentioned, scheduling optimal deployment configurations for application components is out of the scope of our work, given that it is an entire research topic on its own. With this in mind, our architecture will be tailored to accommodate an additional layer which performs resource scheduling. We envision this layer as an edge-enabled application, composed by multiple components, that will employ decentralized scheduling algorithms to determine the aforementioned optimal deployments.

To enable layer, the resource management mechanism will provide two operations: a subscription operation which enables the resource scheduling mechanism to be notified whenever an application component changes its location due to the alerts or failures, and a operation for querying available resources and issuing deployment configurations.

Lastly, we have the **resource location and discovery** mechanism. This mechanism will offer multiple search strategies which, in turn, will be employed by the resource management layer to replace failed components or to satisfy queries arising from the resource scheduling components.

3.3 Evaluation

In order to evaluate our work, we will employ a real-world scenario composed by devices ranging across the different levels of capacity and availability. The devices composing the test scenario consist of: devices in Cloud Environments (e.g. AWS or Azure), devices in the Grid5000 cluster and approximately 20 Raspberry Pis.

We intend to develop two simple solutions that will be representative of common architectures of monitoring systems. The first and most popular approach consists of a centralized controller for tracking and managing the state of devices and components running on them. On the other hand, the second approach consists of a flat decentralized model, meaning that all nodes in the system handle similar amounts of monitoring information. These simple solutions will serve as a baseline for evaluating the benefit of our solution.

In order to compare solutions, we define a set of system and applicational-related metrics. **System metrics** consist of the usage of system resources such as cpu, memory, and bandwidth in each node of the system, followed by the number of required control messages to maintain the overlay.

Application metrics are related to the monitoring infrastructure running atop the overlay. The first metric to consider is *cost*, that consists in the relation between the number of messages sent and the value of the information. Following, we have *information freshness*, that tracks the timeliness of the information each node has of the system, and, finally, *information precision*, which represents the difference between the obtained monitoring data and the real status of the device / applications running on it.

3.4 Scheduling

In this section we outline the identified tasks and the proposed work plan for the remaining of this thesis.

1. **Topology Management** (2/3/2020 - 4/4/2020)
   a) Devise and develop the overlay algorithm which establishes the hierarchical topology (2/3/2020 - 31/3/2020)
   b) Validate the devised algorithm (21/3/2020 - 4/4/2020)
2. **Resource Monitoring** (2/4/2020 - 23/5/2020)
   a) Define the metrics to collect, and devise or adapt a monitoring probe which extracts these metrics from deployed components (2/4/2020 - 8/4/2020)
   b) Implement an adapt aggregation techniques for these metrics based on the overlay structure (10/4/2020 - 1/5/2020)
   c) Create or adapt an alerting solution which analyzes the aggregation results, detects anomalies on running components (e.g., sub-optimal performance) and emits alerts (1/5/2020 - 14/5/2020)
   d) Test the resource monitoring mechanism (26/4/2020 - 23/5/2020)
3. **Resource Location and Discovery** (22/5/2020 - 6/10/2020)
   a) Implement search algorithms which serve as tools for finding certain resources in the system (22/5/2020 - 7/6/2020)
   b) Test the implemented search algorithms (31/5/2020 - 10/6/2020)
4. **Resource Management** (7/6/2020 - 3/7/2020)

a) Implement the handlers for the alerts emitted by the monitoring mechanism (7/6/2020 - 21/6/2020)

b) Implement subscription operations which, in turn, will be employed by the aforementioned resource scheduler (21/6/2020 - 30/6/2020)

c) Test the resource management mechanism (27/6/2020 - 3/7/2020)

5. **Evaluation** (1/7/2020 - 15/8/2020)

a) Develop the centralized and flat solutions to serve as baseline for evaluating the performance of the system (1/7/2020 - 13/7/2020)

b) Develop simple mock-ups of edge-enabled applications (14/7/2020 - 21/7/2020)

c) Perform an experimental assessment of the solution in a realistic test bed, and compare the performance between the developed system and the baseline solutions (22/7/2020 - 15/8/2020)

6. **Thesis document** (1/7/2020 - 23/9/2020)

a) Review related work (1/7/2020 - 31/7/2020)

b) Writing the document (3/3/2020 - 23/9/2020)

c) Reviewing the written document (3/9/2020 - 23/9/2020)

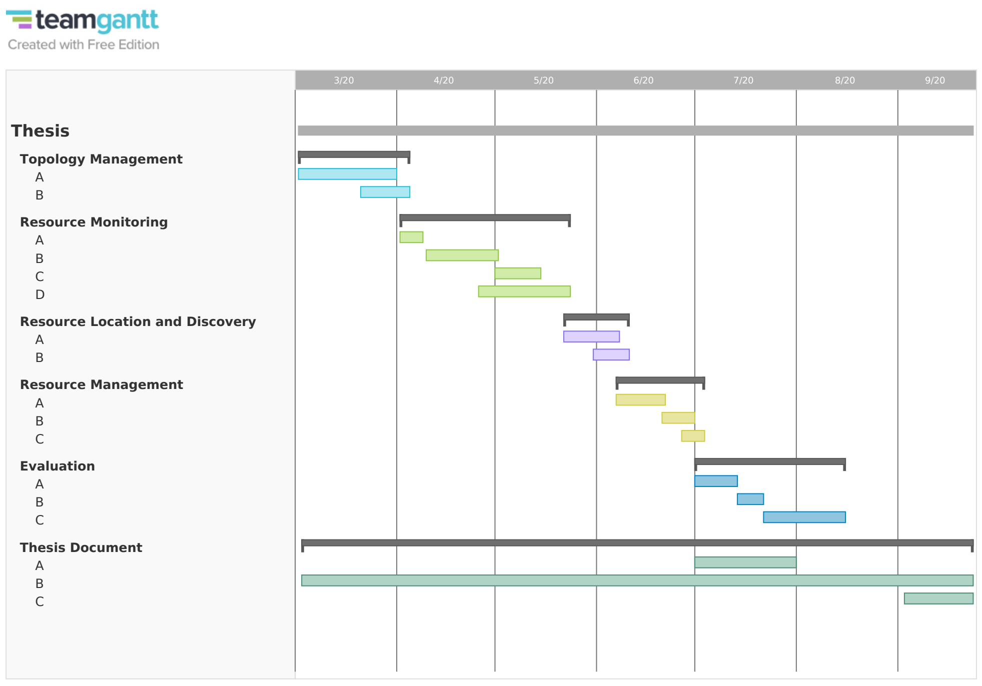

Figure 3.2: Gantt chart illustrating the work plan

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and et al. "A View of Cloud Computing." In: *Commun. ACM* 53.4 (Apr. 2010), 50–58. issn: 0001-0782. doi: 10.1145/1721654.1721672. url: https://doi.org/10.1145/1721654.1721672.

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

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog computing and its role in the internet of things." In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16.

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

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

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

[7] *Catch Pokemon in the Real World with Pokemon GO!* url: https://www.pokemongo.com/en-us/.

[8] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. "Making Gnutella-like P2P Systems Scalable." In: *Computer Communication Review* 33.4 (2003), pp. 407–418. issn: 01464833. doi: 10.1145/863997.864000.

[9] Cloudviz. *cloudviz/agentless-system-crawler*. 2019. url: https://github.com/cloudviz/agentless-system-crawler.

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

[11] P. Costa and J. Leitao. "Practical Continuous Aggregation in Wireless Edge Environments." In: Oct. 2018, pp. 41–50. doi: 10.1109/SRDS.2018.00015.

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

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

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. “Dynamo: amazon’s highly available key-value store.” In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[15] *docker stats*. 2020. URL: https://docs.docker.com/engine/reference/commandline/stats/.

[16] P. Druschel and A. Rowstron. “PAST: a large-scale, persistent peer-to-peer storage utility.” In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. 2001, pp. 75–80. DOI: 10.1109/HOTOS.2001.990064.

[17] *Empowering App Development for Developers*. URL: https://www.docker.com/.

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

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

[20] Google. *google/cadvisor*. 2020. URL: https://github.com/google/cadvisor.

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

[22] *Gtk-Gnutella*. 2019. URL: https://sourceforge.net/projects/gtk-gnutella/.

[23] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Van Renesse. “Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead.” In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 160–169.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

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

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010.

[27] *Infrastructure for container projects*. URL: https://linuxcontainers.org/.

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

[29] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. "Gossip-based peer sampling." In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es.

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

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

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

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

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

[35] J. Leitao, L. Rosa, and L. Rodrigues. "Large-scale peer-to-peer autonomic monitoring." In: *2008 IEEE Globecom Workshops*. IEEE. 2008, pp. 1–5.

[36] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. "X-bot: A protocol for resilient optimization of unstructured overlay networks." In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2175–2188.

[37] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. "Towards Enabling Novel Edge-Enabled Applications." In: 732505 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989.

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

[39] C. Li, Y. Xue, J. Wang, W. Zhang, and T. Li. "Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management." In: *ACM Comput. Surv.* 51.2 (Apr. 2018). ISSN: 0360-0300. DOI: 10.1145/3154815. URL: https://doi.org/10.1145/3154815.

[40] J. Li, J. Stribling, T. Gil, R. Morris, and M. Kaashoek. "Comparing the Performance of Distributed Hash Tables Under Churn." In: Mar. 2004. DOI: 10.1007/978-3-540-30183-7_9.

[41] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. "MON: On-Demand Overlays for Distributed System Management." In: *WORLDS*. Vol. 5. 2005, pp. 13–18.

[42] Y. Mao, C. You, J. Zhang, K. Huang, and K. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective." In: *IEEE Communications Surveys & Tutorials* PP (Aug. 2017), pp. 1–1. DOI: 10.1109/COMST.2017.2745201.

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

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

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

[46] G. Peng. "CDN: Content distribution network." In: *arXiv preprint cs/0411069* (2004).

[47] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri. "Evaluation of Docker containers based on hardware utilization." In: *2015 International Conference on Control Communication & Computing India (ICCC)*. IEEE. 2015, pp. 697–700.

[48] Prometheus. *From metrics to insight*. URL: https://prometheus.io/.

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

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

[51] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. "Scribe: The Design of a Large-Scale Event Notification Infrastructure." In: *Networked Group Communication*. Ed. by J. Crowcroft and M. Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 30–43. ISBN: 978-3-540-45546-2.

[52] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. "Omega: flexible, scalable schedulers for large compute clusters." In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

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

[54] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. ISSN: 2372-2541. DOI: 10.1109/JIOT.2016.2579198.

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

[56] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications." In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

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

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

[59] "Topology Management for Unstructured Overlay Networks." In: *Technical University of Lisbon* (2012).

[60] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. "Apache Hadoop YARN: yet another resource negotiator." In: *SOCC '13*. 2013.

[61] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. "Cloudlets: Bringing the Cloud to the Mobile User." In: *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*. MCS '12. Low Wood Bay, Lake District, UK: Association for Computing Machinery, 2012, pp. 29–36. ISBN: 9781450313193. DOI: 10.1145/2307849.2307858. URL: https://doi.org/10.1145/2307849.2307858.

[62] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-scale cluster management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[63] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. “Osmotic computing: A new paradigm for edge/cloud integration.” In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83.

[64] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. “ENORM: A framework for edge node resource management.” In: *IEEE transactions on services computing* (2017).

[65] P. Yalagandula and M. Dahlin. “A Scalable Distributed Information Management System.” In: *SIGCOMM Comput. Commun. Rev.* 34.4 (Aug. 2004), 379–390. issn: 0146-4833. doi: 10.1145/1030194.1015509. url: https://doi.org/10.1145/1030194.1015509.

[66] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. “Tapestry: A Resilient Global-Scale Overlay for Service Deployment.” In: *IEEE Journal on Selected Areas in Communications* 22 (July 2003). doi: 10.1109/JSAC.2003.818784.