
**Nuno Morais**

Master of Science

# MicromanEdge: a monitoring and orchestration protocol for microservices in Edge Environments

Dissertação para obtenção do Grau de Mestre em

**Engenharia Informática**

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

Júri

| | |
|---|---|
| Presidente: | Name of the committee chairperson |
| Arguente: | Name of a raporteur |
| Vogal: | Yet another member of the committee |


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**December, 2019**

# Resumo

Lorem ipsum em Português.

**Palavras-chave:** Palavras-chave (em Português) ...

# Abstract

Lorem ipsum in english.

**Keywords:** Keywords (in English) ...

# Índice

Capítulo 1

# Introduction

## 1.1 Context

Nowadays, the Cloud Computing paradigm is the standard for development, deployment and management of services, it has proven to have massive economic benefits that make it very likely to remain permanent in future of the computing landscape. It provides the illusion of unlimited resources available to services, and has changed the way developers, users and businesses rationalize about applications [1].

Currently, most software present in our everyday life such as Google Apps, Amazon, Twitter, among many others is deployed on some form of cloud service. However, currently, the rise in popularity of mobile applications and IoT applications differs from the centralized model proposed by the Cloud Computing paradigm. With recent advances in the IoT industry, it is safe to assume that in the future almost all consumer electronics will play a role in producing and consuming data. As the number of devices at the edge and the data they produce increases rapidly, transporting the data to be processed in the Cloud will become unfeasible.

Systems that require real-time processing of data may not even be feasible with Cloud Computing. When the volume of data increases, transporting the data in real time to a Data Center is impossible, for example, a Boeing 787 will create around 5 gigabytes of data per second [8], and Google's self-driving car generates 1 Gigabyte every second [17], which is infeasible to transport to the DC for processing and responding in real-time.

When all computations reside in the data center (DC), far from the source of the data, problems 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 client to the DC as well as the security aspects that arise from offloading data storage and computation, have directed

us into a post-cloud era where a new computing paradigm emerged, Edge Computing.

Edge computing takes into consideration all the computing and network resources that act as an "edge"along the path between the data source and the DC and addresses the increasing need for supporting interaction between cloud computing systems and mobile or IoT applications [18]. However, when accounting for all the devices that are external to the DC, we are met by a huge increase in heterogeneity of devices: from Data Centers to private servers, desktops and mobile devices to 5G towers and ISP servers, among others.

## 1.2 Motivation

The aforementioned heterogeneity implies that there is a broad spectrum of computational, storage and networking capabilities along the edge of the network that can be leveraged upon to perform computations that rely on the individual characteristics of the devices performing the tasks, which can vary from from generic computations to aggregation, summarization, and filtering of data. [11]

There have been efforts to move computation towards the Edge of the network, Fog Computing [20], which is an extension of cloud computing from the core of the network to the edge of the network, has shown to benefit web application performance [23]. Additionally, Content Distribution Networks [] and Cloudlets [] are an extension of this paradigm and are extensibly used nowadays.

Recently, and hand in hand with the trend of cloud decentralization, applications have strayed away from their monolithic application architecture towards splitting in smaller, cohesive, and independent services (microservices). Microservices interact using messages [6], and may cooperate to perform applicational tasks. Specializing services provides many advantages for businesses: each service may be independently built, versioned, and maintained, in addition to the load-balancing, scalability and fault tolerance that arise from employing a decentralized architecture [14].

However, the study state of the art reveals that there is a lack of unified infrastructure that provides the necessary scalability along with resource location and monitoring in to efficiently support dynamic service deployment and management in the Edge. An infrastructure capable of performing the aforementioned tasks successfully has strong applicability in todays world: from commercial purposes (for example, Cloud providers that look to better employ IoT devices), or Smart Cities/Countries.

Given this, p2p protocols must be devised that provide exact resource location and monitoring while providing resource locality according the device distribution in the Edge environment. A common approach to leverage on device heterogeneity is to federate devices in hierarchical topologies. Hierarchical topologies handle the churn and network instability that arises when relying on devices that do not have the same infrastructure as those in Data Centers. Consequently, the hierarchy be used to employ efficient aggregation protocols towards device and service monitoring.

## 1.3 Expected Contributions

The expected contributions from this work are two fold:

- Devise a framework built on top of a membership algorithm that employs a hierarchical topology which reflects the Edge device distribution. Then, exploit the topology to perform aggregation over device and service status. Finally, utilize the aggregation results to efficiently deploy services in Edge environments.
- Perform experimental comparisons with other resource location protocols under various types of environments, namely scenarios where we vary network instability, device heterogeneity, and application load.

## 1.4 Thesis structure

This document is structured in the following manner:

**Chapter 2** focuses on the related work, first and second sections cover P2P systems and the different types of topology management protocols, with an emphasis on structured and self-adapting overlays, third section studies the different types of aggregation and popular implementations for each aggregation type. Fourth section addresses resource discovery and how to perform efficient searches over networks composed by a large number of devices. Finally, last section discusses recent approaches towards enabling Edge Computing along with discussion about Fog, Mist and Osmotic Computing.

**Chapter 3** further explains the proposed contribution along with the work plan for the remainder of the thesis.

Capítulo 2

# Related Work

## 2.1 Context

The following chapter provides context about to the challenge we attempt to solve. We present the sub-challenges identified towards performing management of microservices in the edge of the network.

First, **microservice networking**: microservices need to cooperate towards solving applicational tasks, as such, peers in the system need to be integrated in an efficient abstraction layer (e.g. overlay) that allows them to find each other and communicate. This raises an old challenge in P2P computing: how can peers organize themselves in the network such that they can find resources (either other peers, services or even computing power) in an efficient way? It is important to notice that the edge environment is composed by lots of sub-networks composed of heterogenous devices concurrently entering and leaving the network, which is a hard scenario in **resource location systems**, particularly the underlying **topology management** challenge, because the overlay needs to adapt to the underlying network to remain efficient.

Given this, section 2.2 of this chapter provides context about **topology management**, particularly the main categories of topologies, how they can be evaluated, and a discussion about their applicability in edge environments. Second section, 2.3 studies **resource location architectures**, which leverage on the different types of topologies studied in 2.2 to index resources in the system. For each type of architecture we discuss their differences and present popular implementations in the state of the art.

The next challenge is **maintaining quality of service** of microservices. When attempting to maintain quality of service we need to perform **monitoring** of device and service status, particularly of devices in the edge environment. Section 2.4 covers popular techniques used in tracking device and service status, we particularly study which metrics

to collect to aid in pinpointing causes of QoS degradation, and how to perform failure detection in edge systems, in the same section we discuss popular implementations of monitoring systems.

Then, when federating and tracking massive amounts of monitoring data, transmitting and storing it becomes a limitation in the scalability of the system. To circumvent this, that data needs to be **aggregated**. Section 2.5 studies aggregation, aggregation consists in the process of combining several numeric values into one single representative. We discuss the different types of aggregation, and how they can be applied towards maintaining quality of service.

Lastly, section 2.6 studies how the aggregated results can be used to perform microservice management and deploymen. Namely how to orchestrate microservices such that the computation is offloaded to the Edge. We discuss popular paradigms such as Fog Computing and Osmotic Computing and how they compare to Edge Computing. Lastly, we discuss approaches towards implementing elastic computing in Edge environments.

## 2.2 Topology Management

Given that the main challenge to solve is to provide a platform which enables microservice deployment and management in Edge devices, it is imperative that services are able to find the resources they need (either other services or peers in the system) to meet their requests. For this, peers need implement a **resource location system**.

Resource location systems are one of the most common applications of the P2P paradigm. In these systems, a participant provided with a resource descriptor is able to query peers and obtain an answer to the location (or absence) of that resource in the system within a reasonable amount of time. There are many types of queries a resource location system may support, we present some of them:

1. **Exact Match queries** specify the resource to search by the value of a specific attribute (for example, a hash of the value).

2. **Keyword queries** employ one or more keywords (or tags) combined with logical operators to describe resources (e.g. "pop", "rock", "pop and rock"...). These queries return a list of resources and peers that own a resource whose description matches the keyword(s).

3. **Range queries** retrieve all resources whose value is contained in a given interval (e.g. "movies with 100 to 300 minutes of duration"). These queries are especially applied in databases.

4. **Arbitrary queries** are queries that aim to find a set of nodes or resources that satisfy one or more arbitrary conditions, a possible example of an arbitrary query is looking for a set resources with a certain size or format.

5.

Disseminating the previously mentioned queries in an efficient way through the overlay is a challenge in P2P resource location systems, as such, there have been devised many **dissemination strategies** whose applicability depends on the applicational requirements and system capabilities. There are two main types of dissemination, **flooding** and **walks**.

When **flooding**, peers eagerly forward queries to other peers in the system , the objective of flooding is to contact a certain number of distinct peers in the system that may have the desired 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, it is not scalable and has lots of message redundancy.

**Flooding with limited horizon** minimizes the message redundancy overhead by attaching a TTL to messages that limits the number of times a message can be retransmitted. With limited horizon flooding, there is no guarantee that if a node performs a query to an existing resource in the system,that resource is found.

**Walks** are a dissemination strategy that attempts to minimize the communication overhead that accompanies flooding. Instead of peers forwarding messages to multiple peers, walks are forwarded one peer at a time throughout the system. How walks are propagated in the system dictates the type of walk being used: if walks are propagated randomly, they are said to be **random walks**.

Conversely, walks may take a biased paths in the system based on information accumulated by peers, this is called a **guided random walk**. Guided walks implement the concept of routing indices that allow nodes to forward queries to neighbors that are more likely to have answers [0]. Another approach to bias walks is to use bloom filters, which is a space-efficient probabilistic data structure that supports set membership queries.

Throughout the years 3 popular architectures emerged that are commonly towards indexing resources in a distributed system:

#### 2.2.0.1 Centralized Architectures

**Centralized** architectures rely on one (or a group of) centralized peers that index all resources in the system. This type of architecture greatly reduces the complexity of systems, as peers only need to contact a subset of nodes to locate resources. However, their scalability is limited, due to the centralized point of failure.

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). Some systems use a combination of architectures with success: file sharing systems like Napster and BitTorrent [3] are two examples of hybrid resource location systems that lasted the test of time.

Because centralized architectures have limited scalability, purely centralized architectures cannot be applied in large scale Edge environments. However, there are many

ways that a hybrid architecture can be applied to Edge computing: since the failure rate of a single DC is low, if we assume a system composed by multiple DCs, they may act as a reliable failover for whenever Edge devices are partitioned of fail, DCs can act as an entrypoint for fully distributed systems, or finally, act as an optimization reference point for peers to organize themselves. Conversely, if some of those DCs fail, edge devices may also act as failover.

#### 2.2.0.2 Distributed Hash tables

**Distributed Hash Tables** (DHTs) contrast with centralized servers, where the index distribution is split among peers in the system. In a DHT, peers are assigned uniformly distributed IDs using hash functions, then, peers employ a global coordination mechanism to restrict their neighboring relations (usually called routing tables) such that the resulting overlays commonly consist rings, hypercubes, among others.

Peers maintain routing tables to forward messages in the system, such that they can contact any other participant in a bounded number of steps, where the bound (usually logarithmic) is dictated by the topology . Finally, using the same hash functions to map resources (files, multimedia, messages, etc.) to the peer identifier space, and assigning a key-space interval to each peer, peers can store and find any resource in a bounded number of steps (**exact resource location**).

One particular type of DHT that is commonly employed in in small to medium sized storage solutions is the One-Hop DHT, nodes in a one-hop DHT have **Full membership** of the system and consequently, can perform lookups in O(1) time and message complexity. Kademlia [13] and Amazon's Dynamo [5] are widely used implementations of one-hop DHTs. However, full membership solutions have scalability problems due to memory constrains message volume necessary to maintain the membership information up-to-date. Finally, maintaining full membership is costly in the presence churn (participants entering and leaving the system concurrently). The accumulation of these factors make full-membership solutions impractical in Edge environments, because they usually present high churn (need citation)

Given this, the usual approach to building a DHT is through **partial membership** systems, which rely on some membership mechanism that restricts neighboring relations that are used to perform communication. DHTs with partial membership are attractive because they provide exact resource location while maintaining very little membership information (typically 1% of the peers), .

DHTs have been extensibly used to support many large scale services (publish-subscribe, file sharing, among others) and are especially used in Cloud-based environments. However, there have been attempts to apply DHTs towards Edge Computing. Common limitations that arise from this is that the common flat design goes against the device heterogeneity of Edge Environments. Furthermore, given that devices in those environments have lower computational power and weaker connectivity, devices in the edge may be a

bottleneck to the system.

Following we present some popular implementations of relevant DHT's along with a discussion on their applicability towards Edge environments:

**Chord** [19] is a distributed lookup protocol that addresses the need to locate the node that stores a particular data item, it specifies how to find the locations of keys, how nodes recover from failures, and how nodes join the system. Chord assigns each node and key an m-bit identifier that is uniformly distributed in the id space (peers receive roughly the same number of keys). Peers are ordered by identifier in a clockwise circle, then, any key $k$ 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 circle. This means that whenever the finger table is up-to-date, lookups only take logarithmic time to finish.

Chord, although provides the best trade-off between bandwidth and lookup latency [12], however, chord presents some limitations: peers do not learn routing information from incoming requests and links have no correlation to latency or traffic locality.

Chord is a basis for lots of work: Cyclone [2] is a hierarchical version of Chord provides that constructs a hierarchy by splitting the ID space into a PREFIX and SUFIX. The PREFIX provides intra-cluster identity, whereas the SUFIX is used towards creating clusters of nodes. Routing procedures are executed in lower rings and move up the hierarchy. Hieras [22] uses a binning scheme according the underlay topology to group peers into smaller rings. The lower the ring, the smaller the average link latency. Routing is similar to Cyclone. Crescendo [9] splits the ID range into domains (similar to DNS), where nodes in leaf-domains form Chord rings, then nodes merge rings by applying rules such that rings in different domains can communicate. The resulting routing table and the routing procedures in Crescendo are similar to chord.

**Pastry** [15] is a DHT that assigns a 128-bit node identifier (nodeId) to each peer in the system. The nodes are randomly generated thus uniformly distributed in the 128-bit nodeId space. Nodes store values whose keys are also distributed in the nodeId space. Key-value pairs are stored among nodes that are numerically closest to the key. This is accomplished by: 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, Pastry routes messages towards the numerically closest nodeId. This routing technique accomplishes routing in O(log N), where N is the number of Pastry nodes in the system. This protocol has been widely used and tested in applications such as Scribe [16] and PAST [7]. Limitations from using Pastry arise from the use of a numeric distance function towards the end of the routing process, which creates discontinuities at some node ID values, and complicates attempts at formal analysis of worst case behavior.

**Kademlia** [13] is a DHT with provable consistency and performance in a fault-prone environment. Kademlia nodes are assigned 160-bit identifiers uniformly distributed in the ID space. Peers route queries and locate nodes by employing a novel **XOR-based**

**distance** function that is symmetric and unidirectional. Each node in Kademlia is a router whose routing tables consist of shortcuts to peers whose XOR distance is between $2^i$ by $2^{i+1}$ in the ID space. Intuitively, and similar to Pastry, "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) 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** [10] exploits increased memory usage and constant background communication to achieve O(1) lookup time and message complexity. Kelips nodes are split in $k$ affinity groups split in the intervals $[0,k-1]$ of the ID space, thus, with $n$ nodes in the system, each affinity group contains $\frac{n}{k}$ peers. Each node stores a partial set of nodes contained in the same affinity group and a small set of nodes lying in foreign affinity groups. Through increased communication cost by employing Gossip protocols and memory consumption ($O(\sqrt{n})$ assuming a proportional number of files and peers in the system and a fixed view of nodes in foreign affinity groups), Kelips achieves O(1) time and message complexity in lookups. However, system scalability is limited when compared to Pastry, Chord or Kademlia.

**Tapestry** [21] Is a DHT similar to pastry where messages are incrementally forwarded to the destination digit by digit (e.g. ***8 -> **98 -> *598 -> 4598). Lookups have logb(n) time complexity where b is the base of the ID space. A system with n nodes has a resulting topology composed of n spanning trees, where each node is the root of its own tree. Because nodes assume that the preceding digits all match the current node's suffix, it only needs to keep a constant size of entries at each route level. Thus, nodes contain entries for a fixed-sized neighbor map of size b.log(N).

### 2.2.1 Hybrid Approaches

**Curiata**

**Build One Get One Free**

#### 2.2.1.1 Discussion

## 2.3 Resource Location and Discovery

Given that the main challenge to solve is to provide a platform which enables microservice deployment and management in Edge devices, it is imperative that services are able to find the resources they need (either other services or peers in the system) to meet their requests. For this, peers need implement a **resource location system**.

Resource location systems are one of the most common applications of the P2P paradigm. In these systems, a participant provided with a resource descriptor is able to query

peers and obtain an answer to the location (or absence) of that resource in the system within a reasonable amount of time. There are many types of queries a resource location system may support, we present some of them:

1. **Exact Match queries** specify the resource to search by the value of a specific attribute (for example, a hash of the value).

2. **Keyword queries** employ one or more keywords (or tags) combined with logical operators to describe resources (e.g. "pop", "rock", "pop and rock"...). These queries return a list of resources and peers that own a resource whose description matches the keyword(s).

3. **Range queries** retrieve all resources whose value is contained in a given interval (e.g. "movies with 100 to 300 minutes of duration"). These queries are especially applied in databases.

4. **Arbitrary queries** are queries that aim to find a set of nodes or resources that satisfy one or more arbitrary conditions, a possible example of an arbitrary query is looking for a set resources with a certain size or format.

5.

Disseminating the previously mentioned queries in an efficient way through the overlay is a challenge in P2P resource location systems, as such, there have been devised many **dissemination strategies** whose applicability depends on the applicational requirements and system capabilities. There are two main types of dissemination, **flooding** and **walks**.

When **flooding**, peers eagerly forward queries to other peers in the system , the objective of flooding is to contact a certain number of distinct peers in the system that may have the desired 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, it is not scalable and has lots of message redundancy.

**Flooding with limited horizon** minimizes the message redundancy overhead by attaching a TTL to messages that limits the number of times a message can be retransmitted. With limited horizon flooding, there is no guarantee that if a node performs a query to an existing resource in the system,that resource is found.

**Walks** are a dissemination strategy that attempts to minimize the communication overhead that accompanies flooding. Instead of peers forwarding messages to multiple peers, walks are forwarded one peer at a time throughout the system. How walks are propagated in the system dictates the type of walk being used: if walks are propagated randomly, they are said to be **random walks**.

Conversely, walks may take a biased paths in the system based on information accumulated by peers, this is called a **guided random walk**. Guided walks implement the concept of routing indices that allow nodes to forward queries to neighbors that are more

likely to have answers [0]. Another approach to bias walks is to use bloom filters, which is a space-efficient probabilistic data structure that supports set membership queries.

Throughout the years 3 popular architectures emerged that are commonly towards indexing resources in a distributed system:

#### 2.3.0.1 Centralized Architectures

**Centralized** architectures rely on one (or a group of) centralized peers that index all resources in the system. This type of architecture greatly reduces the complexity of systems, as peers only need to contact a subset of nodes to locate resources. However, their scalability is limited, due to the centralized point of failure.

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). Some systems use a combination of architectures with success: file sharing systems like Napster and BitTorrent [3] are two examples of hybrid resource location systems that lasted the test of time.

Because centralized architectures have limited scalability, purely centralized architectures cannot be applied in large scale Edge environments. However, there are many ways that a hybrid architecture can be applied to Edge computing: since the failure rate of a single DC is low, if we assume a system composed by multiple DCs, they may act as a reliable failover for whenever Edge devices are partitioned of fail, DCs can act as an entrypoint for fully distributed systems, or finally, act as an optimization reference point for peers to organize themselves. Conversely, if some of those DCs fail, edge devices may also act as failover.

#### 2.3.0.2 Distributed Hash tables

**Distributed Hash Tables** (DHTs) contrast with centralized servers, where the index distribution is split among peers in the system. In a DHT, peers are assigned uniformly distributed IDs using hash functions, then, peers employ a global coordination mechanism to restrict their neighboring relations (usually called routing tables) such that the resulting overlays commonly consist rings, hypercubes, among others.

Peers maintain routing tables to forward messages in the system, such that they can contact any other participant in a bounded number of steps, where the bound (usually logarithmic) is dictated by the topology . Finally, using the same hash functions to map resources (files, multimedia, messages, etc.) to the peer identifier space, and assigning a key-space interval to each peer, peers can store and find any resource in a bounded number of steps (**exact resource location**).

One particular type of DHT that is commonly employed in in small to medium sized storage solutions is the One-Hop DHT, nodes in a one-hop DHT have **Full membership**

of the system and consequently, can perform lookups in O(1) time and message complexity. Kademlia [13] and Amazon's Dynamo [5] are widely used implementations of one-hop DHTs. However, full membership solutions have scalability problems due to memory constrains message volume necessary to maintain the membership information up-to-date. Finally, maintaining full membership is costly in the presence churn (participants entering and leaving the system concurrently). The accumulation of these factors make full-membership solutions impractical in Edge environments, because they usually present high churn (need citation)

Given this, the usual approach to building a DHT is through **partial membership** systems, which rely on some membership mechanism that restricts neighboring relations that are used to perform communication. DHTs with partial membership are attractive because they provide exact resource location while maintaining very little membership information (typically 1% of the peers), .

DHTs have been extensibly used to support many large scale services (publish-subscribe, file sharing, among others) and are especially used in Cloud-based environments. However, there have been attempts to apply DHTs towards Edge Computing. Common limitations that arise from this is that the common flat design goes against the device heterogeneity of Edge Environments. Furthermore, given that devices in those environments have lower computational power and weaker connectivity, devices in the edge may be a bottleneck to the system.

Following we present some popular implementations of relevant DHT's along with a discussion on their applicability towards Edge environments:

**Chord** [19] is a distributed lookup protocol that addresses the need to locate the node that stores a particular data item, it specifies how to find the locations of keys, how nodes recover from failures, and how nodes join the system. Chord assigns each node and key an m-bit identifier that is uniformly distributed in the id space (peers receive roughly the same number of keys). Peers are ordered by identifier in a clockwise circle, then, any key $k$ 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 circle. This means that whenever the finger table is up-to-date, lookups only take logarithmic time to finish.

Chord, although provides the best trade-off between bandwidth and lookup latency [12], however, chord presents some limitations: peers do not learn routing information from incoming requests and links have no correlation to latency or traffic locality.

Chord is a basis for lots of work: Cyclone [2] is a hierarchical version of Chord provides that constructs a hierarchy by splitting the ID space into a PREFIX and SUFIX. The PREFIX provides intra-cluster identity, whereas the SUFIX is used towards creating clusters of nodes. Routing procedures are executed in lower rings and move up the hierarchy. Hieras [22] uses a binning scheme according the underlay topology to group peers into smaller rings. The lower the ring, the smaller the average link latency. Routing is similar

to Cyclone. Crescendo [9] splits the ID range into domains (similar to DNS), where nodes in leaf-domains form Chord rings, then nodes merge rings by applying rules such that rings in different domains can communicate. The resulting routing table and the routing procedures in Crescendo are similar to chord.

**Pastry** [15] is a DHT that assigns a 128-bit node identifier (nodeId) to each peer in the system. The nodes are randomly generated thus uniformly distributed in the 128-bit nodeId space. Nodes store values whose keys are also distributed in the nodeId space. Key-value pairs are stored among nodes that are numerically closest to the key. This is accomplished by: 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, Pastry routes messages towards the numerically closest nodeId. This routing technique accomplishes routing in O(log N), where N is the number of Pastry nodes in the system. This protocol has been widely used and tested in applications such as Scribe [16] and PAST [7]. Limitations from using Pastry arise from the use of a numeric distance function towards the end of the routing process, which creates discontinuities at some node ID values, and complicates attempts at formal analysis of worst case behavior.

**Kademlia** [13] is a DHT with provable consistency and performance in a fault-prone environment. Kademlia nodes are assigned 160-bit identifiers uniformly distributed in the ID space. Peers route queries and locate nodes by employing a novel **XOR-based distance** function that is symmetric and unidirectional. Each node in Kademlia is a router whose routing tables consist of shortcuts to peers whose XOR distance is between $2^i$ by $2^{i+1}$ in the ID space. Intuitively, and similar to Pastry, "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) 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** [10] exploits increased memory usage and constant background communication to achieve O(1) lookup time and message complexity. Kelips nodes are split in $k$ affinity groups split in the intervals $[0,k-1]$ of the ID space, thus, with $n$ nodes in the system, each affinity group contains $\frac{n}{k}$ peers. Each node stores a partial set of nodes contained in the same affinity group and a small set of nodes lying in foreign affinity groups. Through increased communication cost by employing Gossip protocols and memory consumption (O($\sqrt{n}$) assuming a proportional number of files and peers in the system and a fixed view of nodes in foreign affinity groups), Kelips achieves O(1) time and message complexity in lookups. However, system scalability is limited when compared to Pastry, Chord or Kademlia.

**Tapestry** [21] Is a DHT similar to pastry where messages are incrementally forwarded to the destination digit by digit (e.g. ***8 -> **98 -> *598 -> 4598). Lookups have logb(n) time complexity where b is the base of the ID space. A system with n nodes has a resulting topology composed of n spanning trees, where each node is the root of its own tree.

Because nodes assume that the preceding digits all match the current node's suffix, it only needs to keep a constant size of entries at each route level. Thus, nodes contain entries for a fixed-sized neighbor map of size b.log(N).

### 2.3.1 Hybrid Approaches

**Curiata**

**Build One Get One Free**

#### 2.3.1.1 Discussion

## 2.4 Monitoring

## 2.5 Aggregation

A basic requirement towards enabling service deployment in the edge is aggregation. Aggregation consists in

Efficient aggregation protocols can be employ

### 2.5.1 Types of aggregation

### 2.5.2 Relevant aggregation protocols

## 2.6 Offloading computation to the Edge

#### 2.6.0.1 Decentralizing clouds

#### 2.6.0.2 Fog Computing

#### 2.6.0.3 Edge Computing

#### 2.6.0.4 Osmotic Computing

Capítulo 3

# Proposed Solution

To achieve this, we propose to create a new novel algorithm which employs a hierarchical topology that resembles the device distribution of the Edge Infrastructure. This topology is created by assigning a level to each device and leveraging on gossip mechanisms to build a structure resembling a FAT-tree [].

The levels of the tree will be determined by ...undecided... and will the tree be used to employ efficient aggregation and search algorithms. Each level of the tree will be composed by many devices that form groups among themselves, the topology of the groups ...undecided...

The purpose of this algorithm is to allow:

1. Efficient resource monitoring to deploy services on.
2. Offloading computation from the cloud to the Edge and vice-versa through elastic management of deployed services.
3. Service discovery enabled by efficiently searching over large amount of devices
4. Federate large amount of heterogeneous devices and use heterogeneity as an advantage for building the topology.

We plan to research existing protocols (both for topology management and aggregation) and enumerate their trade-offs along with how they behave across different environments. Then, employ a combination of different techniques according to their strengths in a unique way that is tailored for this topology.

## 3.1 Document Structure

Capítulo 4

# Planning

## 4.1 Proposed solution

## 4.2 Scheduling

# Bibliografia


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

[2] M. S. Artigas, P. G. López, J. P. Ahulló e A. F. Skarmeta. "Cyclone: A novel design schema for hierarchical DHTs". Em: *Proceedings - Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*. Vol. 2005. 2005, pp. 49–56. ISBN: 0769523765. DOI: 10.1109/P2P.2005.5.

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

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

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

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin e L. Safina. "Microservices : Yesterday , Today , and Tomorrow". Em: (), pp. 195–196.

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

[8] 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.

[9] P. Ganesan, K. Gummadi e H. Garcia-Molina. "Canon in G major: Designing DHTs with hierarchical structure". Em: *Proceedings - International Conference on Distributed Computing Systems* 24 (2004), pp. 263–272. DOI: 10.1109/icdcs.2004.1281591.

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

[11] J. Leitão, P. Á. Costa, M. C. Gomes e N. M. Preguiça. "Towards Enabling Novel Edge-Enabled Applications". Em: *CoRR* abs/1805.06989 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989.

[12] J. Li, J. Stribling, T. Gil, R. Morris e M. Kaashoek. "Comparing the Performance of Distributed Hash Tables Under Churn". Em: mar. de 2004. DOI: 10.1007/978-3-540-30183-7_9.

[13] P. Maymounkov e D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". Em: *Peer-to-Peer Systems*. Ed. por P. Druschel, F. Kaashoek e A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0.

[14] S. Newman. *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc.", 2015.

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

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

[17] *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.

[18] W. Shi, J. Cao, Q. Zhang, Y. Li e L. Xu. "Edge Computing: Vision and Challenges". Em: *IEEE Internet of Things Journal* 3 (out. de 2016), pp. 1–1. DOI: 10.1109/JIOT.2016.2579198.

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

[20] S. Yi, Z. Hao, Z. Qin e Q. Li. "Fog computing: Platform and applications". Em: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE. 2015, pp. 73–78.

[21] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph e J. Kubiatowicz. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". Em: *IEEE Journal on Selected Areas in Communications* 22 (jul. de 2003). DOI: 10.1109/JSAC.2003.818784.

[22] Zhiyong Xu, Rui Min e Yiming Hu. "HIERAS: a DHT based hierarchical P2P routing algorithm". Em: *2003 International Conference on Parallel Processing, 2003. Proceedings.* 2003, pp. 187–194. DOI: 10.1109/ICPP.2003.1240580.

[23] J. Zhu, D. Chan, M. Prabhu, P. Natarajan e H. Hu. "Improving Web Sites Performance Using Edge Servers in Fog Computing Architecture". Em: mar. de 2013, pp. 320–323. ISBN: 978-1-4673-5659-6. DOI: 10.1109/SOSE.2013.73.

Apêndice

# Appendix 2 Lorem Ipsum

ANEXO I

# ANNEX 1 LOREM IPSUM