
**Nuno Morais**

Master of Science

# Monitoring and deploying services on edge devices

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**

**Monitoring and deploying services on edge devices**



Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

# Resumo

Lorem ipsum em Português.

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

# Abstract

Lorem ipsum in english.

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

# Índice

# Lista de Figuras

# Lista de Tabelas

# Listagens

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 [**10.1145/1721654.1721672**].

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 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 [**finnegan_2013**], and Google's self-driving car generates 1 Gigabyte every second [**datafloq**], 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 [**iot_journal_shi_weisong_and_cao**]. 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. [**DBLP:journals/corr/abs-1805-06989**]

There have been efforts to move computation towards the Edge of the network, Fog Computing [**yi2015fog**], 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 [**Improving_Web_Sites_Performance_Using_Edge_Servers_in_Fog_Computing_Architecture**]. Additionally, Content Distribution Networks [] and Cloudlets [] are an extension of this paradigm and are extensibly used nowadays.

To fully materialize the Edge Computing paradigm, applications need to be split into small services that cooperate to fulfill applicational needs. Furthermore, tools must be developed that allow service and resource discovery as well as service and device monitoring. All of this performed over the inherent scale considered by the infrastructure of the Edge Computing paradigm.

paragrafo com sobre para quem e que isto e (e.g. Google, Amazon, Smart City, Smart Country ?? also porque e que isto e diferente do que ja existe

These tools must be based on protocols that are able to federate all the devices and aggregate massive amounts of device data in order to perform efficient service deployment and management. Finally, they must handle the churn and network instability that arises when relying on devices that do not have the same infrastructure as those in Data Centers.

## 1.3 Expected Contribution

Capítulo 2

# Related Work

## 2.1 Decentralizing clouds

#### 2.1.0.1 Fog Computing

#### 2.1.0.2 Edge Computing

#### 2.1.0.3 Osmotic Computing

## 2.2 Peer to Peer

The Peer to Peer (P2P) paradigm has been extensibly used to implement distributed, scalable, fault-tolerant services, it overcomes limitations such as scalability and fault-tolerance that arise from the client-server model.

Collaboration is the foundation of scalability provided by P2P. Participants of the system (peers) perform tasks that contribute towards a common goal which is beneficial for the overall functioning of the system. Collaboration enables P2P systems that accomplish tasks would otherwise be impossible by an individual server.

In a P2P system peers contribute with a portion of their resources (such as computing, memory or network bandwidth, among others) to other participants as well as consume resources from other peers. This contrasts with the traditional client-server paradigm where the consumption and supply of resources is decoupled. Finally, fault tolerance is achieved by the absence of a centralized point of failure.

There are many types of services that are based on P2P systems, however, most popular types of services build on this paradigm are: resource location, content delivery, cryptocurrency or blockchain, etc. Popular services built on this paradigm are file sharing applications, (e.g. Napster, BitTorrent, Emule and Gnutella), cryptocurrencies (Bitcoin), streaming (Skype), anonymity services (TOR) among others.

One factor that imposes a crucial difference in how these systems behave is the membership information, a participant in the system that knows every other participant in the system is said to have full membership information, the alternative is called partial membership, where a peer is only aware of a partial number of elements in the system.

**Full membership systems are usually employed in small to medium sized storage solutions like on One-Hop distributed hash tables(DHT) e.g. Kademlia [10.1007/3-540-45748-8_5] and Amazon's Dynamo [decandia2007dynamo]. However, full membership solutions have scalability problems due to memory constrains and the increase in workload 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).**

**Partial membership systems rely on some membership mechanism that restricts each peer to only have a few neighboring relations that are used to perform communication (usually through message exchanges) between each other. Similarly to full membership, the number of connections a peer has dictates the scalability of the system, where systems with larger views will have a harder time maintaining them.** The accumulation of the partial views of all peers in the system dictates the topology of the overlay network. Topology management consists in the creation and management of an **overlay network**, which consists in a logical network built on top of another network (usually the internet). Elements of overlays are connected through virtual links that are a combination of one or more underlying physical links.

The way the aforementioned links are organized dictate the type of overlay: if the links are logically organized by some metric, we call it a **structured overlay**. On contrary, when there are no restrictions form the links, we call it an **unstructured overlay**. Services then leverage on the topologies that are tailored as closely as possible towards application requirements to build efficient services.

## 2.3 Resource Location

Resource location is one of the most popular applications of P2P paradigm. Intuitively, any participant provided with a resource descriptor is able to query peers and obtain an answer to the location of that resource in the system.

There are mainly 3 architectures of resource location systems:

**Centralized architectures rely on one (or a group of) centralized peers that index all resources in the system. This greatly reduces the design of the system, however, it is not scalable and, as the name indicates, it has a centralized point of failure. Some systems use a combination of architectures, such as using a centralized service to provide**

**infrastructure for a DHT overlay. Some applications have employed this architecture with success, such as BitTorrent, for serving torrent files through HTTP.**

**Distributed Hash Tables are alternatives to centralized servers, where the index distribution is split among peers in the system. Peers map resources to nodes in the system and employ routing algorithms to transverse the topology in a bounded number of steps. Then, using 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 find any resource in a bounded number of steps (Exact resource location).**

**Unstructured Overlays are the third common approach to resource location systems, in this overlay, each peer will maintain a local index of its own resources and in some cases information of indexes of other peers. Participants then disseminate queries to find sets of nodes that own matching resources.**

#### 2.3.0.1 Distributed Hash tables

DHTs impose restrictions over the neighboring relations that can be established among peers, often based on the identifier of each participant of the system. Usually these identifiers are unique and are independently generated by each peer, then peers employ a global coordination mechanism to tightly control the topology. Examples of common resulting topologies of structured overlays are rings, meshes, hypercubes, among others.

DHT's have been extensibly used to support many large-scale different types of services (publish-subscribe, resource location, monitoring, among others) and are especially used in Cloud-based environments. Their popularity derives from providing the previously mentioned functionality while maintaining very little membership information (typically 1% of the peers). DHT's solve **exact resource location queries**, which consists in being able to find a given node in the system given its ID in a bounded number of steps.

However, a DHT's behavior depends on the correctness of the topology, and when in the presence of high churn or network partitions / failures, the correctness of the topology is harder to maintain as nodes must exchange more messages to ensure that the topology is correct. Additionally, whenever a node fails, DHT's rely on the correctness of the routing infrastructure to replace the failed node, this means that if there is a failure in the routing infrastructure, the DHT may become unrepairable.

Following we present some popular implementations of relevant DHT's along with a discussion with their pros and cons:

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.

Additionally, 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** [**stoica2003chord**] 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.

Additionally, 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** [**stoica2003chord**] 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.

Additionally, 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.

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 [**10.1007/3-540-45546-9_3**] and PAST [**990064**]. 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. ][ **Pastry** [**rowstron2001pastry**] 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 [**10.1007/3-540-45546-9_3**] and PAST [**990064**]. 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. ] **Pastry** [**rowstron2001pastry**] 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 [**10.1007/3-540-45546-9_3**] and PAST [**990064**]. 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.

Peers route queries and locate nodes by employing a 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. ][ **Kademlia** [**10.1007/3-540-45748-8_5**] 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 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. ] **Kademlia** [**10.1007/3-540-45748-8_5**] 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 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 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 n$\frac{}{kpeers.Eachnodestoresapartialsetofnodescontainedinthesa}$

Through increased communication cost by employing Gossip protocols and memory consumption (O($\sqrt{n}$)$assumingaproportionalnumberoffilesandpeersinthesystemandafixedviewofnodesin$

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 n$\frac{}{kpeers.Eachnodestoresapartialsetofnodescontainedinthesa}$

Through increased communication cost by employing Gossip protocols and memory consumption (O($\sqrt{n}$)$assumingaproportionalnumberoffilesandpeersinthesystemandafixedviewofnodesin$

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 n$\frac{}{kpeers.Eachnodestoresapartialsetofnodescontainedinthesa}$

Through increased communication cost by employing Gossip protocols and memory consumption (O($\sqrt{n}$)$assumingaproportionalnumberoffilesandpeersinthesystemandafixedviewofnodesin$

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 $b.(N)$. ][ **Tapestry** [**tapestry**] 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 $b.(N)$. ] **Tapestry** [**tapestry**] 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 $b.(N)$.

surrogate routing

**Viceroy**

**Koala**

### 2.3.1 Unstructured overlays

As previously mentioned, unstructured overlays maintain random topologies that have a low maintenance cost, but at the same time provide limited efficiency and scalability when looking for a specific node in the system. However, if applications do not require exact resource location, unstructured overlays provide attractive characteristics such as fault-tolerance, robustness to churn and flexibility in the format of queries. In the context of resource location systems, unstructured overlays are used to propagate **keyword queries** or **arbitrary queries**.

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

**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.

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 **guided random walk**.

**Hyparview [Hyparview] is a protocol that takes a hybrid approach to membership** There have been many approaches to solve the scalability problem that emerges from

#### 2.3.1.1 Self-adapting overlays

### 2.3.2 Hybrid Approaches

**Curiata**

**Build One Get One Free**

## 2.4 Aggregation

### 2.4.1 Types of aggregation

### 2.4.2 Relevant aggregation protocols

## 2.5 Resource Discovery

## 2.6 Offloading computation to the edge

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

The document is structured in the following manner:

**Chapter 2** focuses on the related work, first section covers the different types of topology management protocols, with an emphasis on random and self-adapting overlays, second section studies the different types of aggregation and popular implementations for each aggregation type. Third section addresses resource discovery and how to perform efficient searches over networks composed by a large number of devices. Finally,fourth 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 4

# Planning

### 4.0.1 Proposed solution

### 4.0.2 Scheduling

Apêndice


# Appendix 2 Lorem Ipsum

Anexo I

# Annex 1 Lorem Ipsum