THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Par

Genc TATO

Lazy and Locality-Aware Building Blocks for Fog Middleware : A Service Discovery Use Case

Thèse présentée et soutenue à RENNES, le 19 décembre 2019
Unité de recherche : Inria Rennes – Bretagne Atlantique
Thèse N° :

Abstract

In the last decade, cloud computing has grown to become the standard deployment environment for most distributed applications. While cloud providers have continuously extended their coverage to different locations worldwide, the distance of their datacenters to the end users still often translates into significant latency and network utilization. With the advent of new families of applications such as virtual/augmented reality and self-driving vehicles, which operate on very low latency, or the IoT, which generates enormous amounts of data, the current centralized cloud infrastructure has shown to be unable to support their stringent requirements. This has shifted the focus to more distributed alternatives such as fog computing. Although the premises of such infrastructure seem auspicious, a standard fog management platform is yet to emerge. Consequently, significant attention is dedicated to capturing the right design requirements for delivering those premises.

In this dissertation, we aim at designing building blocks which can provide basic functionalities for fog management tasks. Starting from the basic fog principle of preserving locality, we design a lazy and locality-aware overlay network called Koala, which provides efficient decentralized management without introducing additional traffic overhead. In order to capture additional requirements which originate from the application layer, we port a well-known microservice-based application, namely Sharelatex, to a fog environment. We examine how its performance is affected and what functionalities the management layer can provide in order to facilitate its fog deployment and improve its performance. Based on our overlay building block and the requirements retrieved from the fog deployment of the application, we design a service discovery mechanism which satisfies those requirements and integrates these components into a single prototype. This full stack prototype enables a complete end-to-end evaluation of these components based on real use case scenarios.

Acknowledgements

Sunny afternoon, sitting on a dock, staring off into the horizon of the northern coast of Bretagne, a gentle breeze caresses my hair, as I hold a notebook and a quill...

More or less that is how I had imagined the moment of writing the acknowledgments on my newly completed manuscript. Little did I know then that writing acknowledgments, like most of the fun activities I have had recently, would have been yet another way to distract myself from dealing with more critical tasks, such as practicing my defense presentation in this case. Oh procrastination, why are you so sweet?! This time, however, it is bittersweet. Like the end of a good movie, I was looking forward to reaching this moment, and yet I feel regretful now that it came. And like at the end of a good movie, I would like to dedicate a moment to all the amazing people who made this journey possible.

I would like to express my gratitude to all the members of the jury: Luciana Arantes, Noël de Palma, Sébastian Monet, Adrian Lebre and Etienne Rivère, for their time and efforts for making this defense day possible. In particular, I would like to thank Noël and Sébastian for accepting the additional task of reading and reviewing my manuscript. Thank you also Adrian for the support in the Discovery project and for being part of my CSID committee. Special thanks to Etienne for hosting me in Louvain-la-neuve and the support and dedication you put into writing those articles.

Of course my deepest appreciations for my two great supervisors, Cédric and Marin. It has been a pleasure and honor working with you. I know that fighting my scepticism and stubbornness sometimes requires super powers. But your guidance, motivation, positivity, and trust in our work and in me, can definitely count as such. Thank you for being there at every step of this journey and always steering me in the right direction.

Thank you Guillaume for giving me the opportunity to join the Myriads team. As the most permanent non permanent, I had the pleasure to meet various generations of the team and share many of my joyful and frustrating moments at work. Thank you David, Yunbo, Clement, Mehdi and Yasmina for making work a happy place and Mathieu for protecting me from the headache of technical issues.

Thank you my close friends for making even the afterwork a happy place: Mada and Bogdi, for your support, positivity and making me feel like I can always rely on you (even when lost somewhere in the Netherlands); Javier, for all the nice conversations and for being there in difficult moments; Amir, for being so caring and reliable (I will never forget that moquette day); Younes, for adopting me and opening the doors of your place; And of course, Cécile, for reminding me to take life easy and for being a relentless trigger of self improvement.

While the classics will always be irreplaceable, the new friends I made became indispensable. I am of course referring to the first aid team: Lorenzo, Paulo and Cristhyne. Giovane, I was extremely lucky to meet you on my return to Rennes. Paulo, I was also lucky you joined the team so late, otherwise I would have never graduated. Cris, where can I start?! I would need again what I have written so far. All I can say is that very few people would have done what you did for me to be here today.

Isma and Anna, no matter how much I love you, treason is not cool! You got one chance to make it right!

Kris, Martin, mom and dad, words will never be enough to express how grateful I am for all you had to endure for me to receive a proper education and reach this day. I deeply love you and I dedicate this work to you.

I am leaving this lab and soon also Rennes, but a part of me will stay forever here and a part of Rennes will be forever with me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Systems

The first modern computers were large, expensive, and only a few organizations could afford them. Due to their scarcity and the lack of means for connecting them, they operated independently. Then, in the mid 80's, two major technological advances began thriving: microprocessors and networks. Over the years, the computing power of microprocessors improved dramatically, while their cost and size dropped significantly. Likewise, advancements in hardware and software layers of the networks enabled data exchange at a very high speed.

Finally, there was communication. At first, this involved computers within a building, or organization, where they were connected in a Local Area Network (LAN). These small networks started gradually interconnecting, by forming increasingly larger networks, resulting in the so called Wide Area Networks (WAN). Eventually, the interconnection of LANs and WANs all over the world gave rise to a global network, the Internet. It was the beginning of the Information Age, a historic period which entirely reshaped industry and most aspects of people's lives [167].

This staggering development was not only the result of simply interconnecting processors in large infrastructures. The real power of such interconnection resided in the way these numerous processors collaborated with each other to accomplish bigger and more complex tasks, that would be otherwise impossible in a reasonable time. The need for such collaboration, gave origin to one of the most central fields in computer science, Distributed Systems. In their book with the same name [157], Tanenbaum and Van Steen define a distributed system as following:

> A distributed system is a collection of independent components (computers) that appears to its users as a single coherent system.

While users are provided with a single coherent interface, behind the scenes, these components can have different roles and interact in various ways. Two of the most common ways of partitioning tasks in distributed systems are the *client-server* and *peer-to-peer* (P2P) models. In a client-server model, tasks are shared between resource requesters, the clients, and resource providers, the servers. Typical examples of this model include the World Wide Web (WWW) and the e-mail. In the P2P model, each component acts at the same time as a client and as a server, by both requesting and providing resources. Common cases of this model include file sharing or Voice over IP (VoIP) systems. Given their potential for solving complex tasks that will arise in the future and their applicability for addressing current demands, distributed systems have been constantly subject of research and an integral part of today's industry.

1.2 Distributed systems in research and industry

It is often difficult to separate how research and industry contribute in the development of a particular field. Very often contributions are intertwined, with research focusing on the big picture, and industry addressing tangible current issues.

Research

Since the early stages research focused on leveraging the potentials of combining the power of multiple processors. A great effort was dedicated to creating communication standards, such as the OSI model and the Internet protocol suite. This was followed by additional protocols and frameworks for facilitating synchronization and collaboration. Programming languages were also enriched with tools to build distributed systems, such as Remote Procedure Calls (RPC), Message Passing Interface (MPI), or the Map-Reduce model. Engaging multiple machines in parallel to perform a certain task had never been easier.

At this point, achieving unprecedented computing performance was a question of horizontally scaling processors. Soon supercomputers performing quadrillions of operations per seconds were a reality. This affected research in many other fields like genetics, climate, space exploration, etc. This exceptional computing power had to be made available to educational or research institutions needing it. The urge to provide computation as utility was manifesting.

Industry

From the first chat applications to the current social media, a plethora of distributed systems came into being thanks to industry. Early companies generally provided client-server applications, where the clients were usually user machines, while servers were machines located on premise. As companies gained increasingly more clients, their computing resources continuously increased. Soon servers became racks, racks filled machine rooms, and in some cases even datacenters. For most of the companies storing, maintaining and upgrading these machines was unsustainable.

On the other hand, new companies started to have high barriers for entering the market, because they required a large initial investment in order to be competitive. Moreover, the demand for computing power was not constant, and therefore it was difficult for companies to plan their investments. As in research, utility computing was a clear subsequent step.

1.3 Utility computing: the Grid and the Cloud

Utility computing relies on the idea of providing computing resources as needed, similar to other utilities like electricity or water. This implied that users needed a way to load their applications or tasks onto the distributed infrastructure in order to execute them, and potentially remove them once done. Given the immense heterogeneity of applications and hardware, this was not straightforward. It is when virtualization technologies proved to be very useful. Virtualization creates an abstraction layer between hardware and software which removes compatibility issues, while providing software with virtually infinite hardware resources. Users can package their full software stack into virtual environments, such as Virtual Machines (VMs) or Containers, and allocate as much resources as needed. When the execution terminates, these environments can be disposed of without affecting the rest of the system.

Around 2004, the concepts of utility computing in supercomputers were introduced in research by the trio Foster, Kesselman, and Tuecke, under the name of Grid computing [54]. Their definition of Grid involved detailed notions regarding authentication, data storage, virtual networks, security provisioning, monitoring, etc. These advancements paved the way for another noted implementation of utility computing, the Cloud.

The Cloud took Grid concepts to the industry. A few large companies which already owned datacenters, such as Amazon or Google, increased their capacities and specialized in providing computing and storage resources to users in a pay-per-use model. Indeed, the Cloud associated a business model to the Grid services, which made virtually unlimited computing power and storage accessible to any other company, for a reasonable price. Not only the cloud has had a tremendous impact in the IT industry, but also it has recently become the de facto computing platform for established companies in various industry fields, and a centerpiece in the success of numerous startups.

Certainly, by giving up their on premise infrastructures, companies relieved themselves from the responsibility of maintaining them and their associated costs. However, oftentimes these infrastructures were located closer to their clients and distributed the overall computation load. By hosting their services in the Cloud, all computation became centralized only in few locations around the world. Given the vast amount of companies turning to Cloud, centralization becomes a real issue which deserves attention.

1.4 Centralization and decentralization

Distributed systems are a story of alternation between centralization and decentralization. Centralization provides simplicity, efficient algorithms, easy management and billing, but also single point of failures and saturation of network and computing resources. On the other hand, decentralization provides robustness and load balancing, but at the cost of highly increased complexity in management and billing. It is often the circumstances deciding which is the most appropriate choice.

In the past, we have already noticed shifts between the two approaches. The first distributed systems were based on the client-server architecture which is highly centralized. However, the limits of this architecture were tested in the early 2000's in the context of file sharing, where the network and central servers were not able to cope with the amount of requests and data transfer. As a result, peer-to-peer architectures emerged, providing a much more efficient decentralized solution. Nevertheless, over the next decade, remarkable advances in networks, computing, and storage took place, which made client-server architectures a viable solution once again. In general, the trend shows that centralization is preferred as long as service usage does not reach the limits of the infrastructure, otherwise decentralization is required until the next technological breakthrough.

1.5 Cloud numbers and limits

Cloud services are one of the most rapidly growing markets in 2019 with a value of $214.3 billion, and is projected to keep a steady growth above 14% until 2022 [60]. According to Cisco's forecast [77], by 2021, 94% of the overall computing will be processed in cloud datacenters. The global annual cloud traffic from 2016 to 2021 will increase 3.3-fold (up to 19.5 ZB), while the data stored in the Cloud will increase 4-fold (up to 2.6 ZB).

While the continuous cloud growth seems far from plateauing, the effect of this growth on the current infrastructure needs to be examined. The end-to-end infrastructure comprises three main elements: the datacenters, the network (links, routers, switches), and the user machines or devices.

Given that datacenters are owned by cloud companies directly benefiting from this growth, their reaction is likely to be more adequate. Indeed, Cisco's report foresees that hyperscale datacenters will dominate the landscape of public cloud by 2021. However, massive datacenters come at extremely high energy consumption. In 2030 consumption of datacenters will account for up to 13% of the global consumption, compared to 1% in 2010 [13]. This is similar to the energy consumption of the USA in 2016 (17%) [6]. Unless there is a profound shift to green energy, the growth of datacenters is not sustainable.

Regarding networks, the response to cloud growth is less immediate. Different network segments are owned and managed by different actors and this creates several issues. Inadequate regulation and competitive dynamics between networking companies can result in sub-optimal

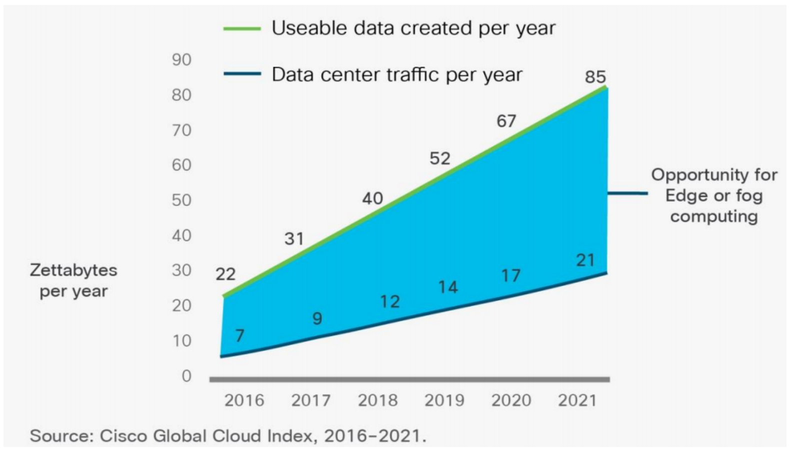


Figure 1.1: Potential IoT effect on Cloud [77].

infrastructures or configurations [39]. Moreover, different from datacenter investments which are centralized, those in fixed network require a capillary intervention in infrastructure, which is expensive and has many technicalities (trenching, government permissions). This means that they cannot be consistent among different companies and locations. Finally, the energy consumption of networking is as steep as that of datacenters, thus it suffers from the same unsustainability issues [13].

As regards user machines, especially mobile devices, they are one of the main promoters of the cloud growth. In the last decade we have witnessed an unprecedented growth in smart mobile devices, reaching up to 5 billion users in 2017 (65% of world population) [79]. This growth has been characterized by groundbreaking progress in computing power, memory and network capabilities, enabling these devices to support whole new categories of applications. From virtual and augmented reality to self-driving cars, the opportunities seem endless. However, many of these applications require very low latencies to operate (below 15 ms [50]). Given the steep increase of cloud traffic and the inability of networks to sustain this growth, it is unattainable for centralized clouds to support these applications.

In addition to the demands for low latency, mobile applications generate an enormous amount of data. According to Cisco, by 2021, 15% of all data will reside in mobile devices, with a high chance to move eventually to the datacenters. Moreover, with the advent of the Internet of Things, the amount of generated data will soar up to 850 ZB per year, where 90% of it is ephemeral (requiring processing, but not necessarily storing). Processing such amounts of data in the centralized cloud would be unrealistic, given that the increase of traffic would be 20 times more than predicted (21 ZB). Even if only the result of such processing was stored (10%), it would still surpass the amount of the overall traffic generated by other services, as shown in Figure 1.1. In order to deal with these limitations, alternative solutions such as edge or fog computing are to be considered.

1.6 Edge and Fog

Edge and Fog both rely on the idea of distributing computing and storage resources across the network, rather than having them in a few central datacenters. The two terms are sometimes used interchangeably. However, a general perception is that Edge focuses on introducing resources as close as possible to the users, at the edge of the network, while Fog is more generic and envisions a distribution of resources in the continuum between the edge and the core of the network.

These concepts aim at addressing the weaknesses of the centralized Cloud. They minimize the latency to the users, while drastically reducing the traffic propagated to the central datacenters in the network core. This does not only support the aforementioned real-time and IoT applications, but also provides a fairer distribution of resources, similar to the initial structure of the Internet. This helps the current infrastructure to handle better the expected increased traffic for cloud services.

The premises of Fog and Edge have converted the two terms into a buzzword among technology enthusiasts. This interest has been reflected similarly in research, especially in the last 5 years. A considerable number of studies have focused on potential architectures and algorithms for delivering on these promises. Many of them focus on domain-specific applications (healthcare [154], self-driving vehicles [73], smart cities [158]), while others focus on particular management aspects, such as scheduling [5], migration [19], load balancing [83], federation [173] etc. However, it is worth noticing that most of the Fog-related studies are often conceptual and rarely come with a prototype [124]. Materializing these ideas into a standard general-purpose Edge/Fog platform is still an ongoing process, which requires further research.

A step toward standardization is to treat infrastructure management as a whole, without building up solutions for specific tasks from the ground up. At first, it is important to address architectural aspects that are common to all management tasks, such as task distribution, scalability, communication primitives, localization, interfaces, and Quality of Service (QoS) management. The design decisions on these aspects must still comply with the premises of Edge/Fog architectures about minimizing latency and traffic generation. Once we establish a framework, the implementation of a particular management task can focus on policies rather than architecture.

Objectives

In this dissertation we go through different steps of addressing architectural aspects of fog management, by providing building blocks that can be used in the implementation of specific management tasks. We aim at tackling this problem in two directions, bottom-up and top-down. While in the bottom-up we analyze management starting from the infrastructure, in the top-down we focus on the application perspective. More specifically, our objectives are the following:

bottom-up

- Starting from the premises of Fog, we aim at identifying the most critical requirements for an efficient infrastructure management

- Based on these requirements, we aim at building an overlay network specialized for a fog environment, which serves as a building block for higher-level management tasks.

top-down

- In order to capture real-life requirements handled in the management layer, we aim at deploying an existing widely-used application in a fog environment.

- In the absence of a real open-source fog-native application, we aim at identifying architectural aspects and approaches to adapt a Cloud application in order to enable its deployment over the Fog.

- We want to learn how effective fog deployment can be for a non-fog-native application and how they can be improved.

meet in the middle

- Based on our overlay network, and the deployed application, we aim at implementing an example of a management task, such as service discovery, which builds upon our overlay and supports application deployment in real-case user scenarios. The overall goal is to provide a fully integrated prototype of management and deployment of an application in a fog infrastructure.

Contributions

Organized similarly to the objectives, our contributions are the following:

bottom up

- We argue that in order to comply with fog principles, an efficient management needs to be uncomplicated, decentralized, preserve locality, and reduce unnecessary traffic, but yet deliver a fast service and be resilient to failure.

- We introduce Koala, an overlay network which provides an implementation of these requirements. Koala is fully decentralized and adopts a simple flat structure, yet it takes in consideration the datacenter-based topology of the fog. It provides locality-awareness using different techniques, including proximity routing and discovery of proximity nodes. Koala minimizes management traffic by removing any maintenance routine. Instead, it includes a lazy mechanism which relies on piggybacking maintenance information on application traffic. Therefore, its maintenance is tied to its usage.

- We provide a simulation-based evaluation of our overlay network under various conditions, using hundreds of thousands of nodes, and compare it to a classical overlay.

top-down

- We argue about the requirements applications must comply with in order to enable their fog deployment without re-implementing them. We propose the microservice-based architecture as a potential implementation of those requirements (inline with the osmotic computing approach).

- We define a taxonomy of microservices based on their state and the impact their replication has on the application deployment. This taxonomy is critical for deciding how and where the microservices should be deployed in the fog.

- We consider the fog deployment of a well-known microservice application, namely Sharelatex. Based on our taxonomy and some additional criteria, such as criticality and usage, we propose a way to split and replicate the application services in various locations.

- We evaluate how this deployment impacts certain application use cases and identify several issues that could have been treated differently if the application was designed with fog deployment in mind.

- In our deployment, most commonly the state of microservices is either fully replicated in different instances, or sharded in such a way that each instance stores a distinct set of stored objects. We observe that finding the best replica or locating the right shard given an object, is a common requirement in such deployment and that there are several advantages if it is handled externally to the application.

integration

- On top of our Koala overlay, we build a microservice discovery mechanism which handles both replicas and shards. While replica selection can be based on several criteria, we showcase low-latency as a relevant criterion in the fog environment. For shard localization we introduce an interface between the application and the discovery mechanism which allows the latter one to identify objects in a request, and redirect the request to the shard containing that object.

- In addition to maintaining a mapping between objects and shards, our system keeps track of the locations of the users who request a particular object. In case their locations are distant from the object's shard, our system migrates the object to a different shard which location is on average closer to the users of that object.

- We integrate our discovery mechanism with ShareLatex in a fully automated fog deployment and show how the system reacts to real-life use cases.

- Finally, we provide some insights and some learned lessons not only for the fog deployment of existing applications, but also for building fog applications from scratch.

Outline

The remainder of this dissertation is organized as follows: Chapter 2 revisits some background concepts and the respective related work. Chapter 3 focuses on analyzing the fog infrastructure, and identifying the requirements for a decentralized management. Chapter 4 describes the details of the Koala overlay network, its architecture, its implementation of requirements identified in the previous chapter, and its evaluation. Chapter 5 describes in details the fog deployment of ShareLatex, discussing the different types of microservices and the criteria used to distribute microservices in various locations in the Fog. Chapter 6 describes the Koala-based service discovery mechanism, its integration with Sharelatex and its evaluation. Finally, in Chapter 7 we draw our conclusions and provide insights into further improvements of this work and related research directions.

Chapter 2

State Of the Art

In this chapter we present the state of the art which serves as a background to our work. We start from the infrastructure layer where we describe the environment and the scope of our work. Then we review the related work in the middleware layer where our contribution is mainly focused. We finally discuss the service layer where our work is put into use for providing an essential service. The chapter is organized as follows: in Section 2.1 we discuss the infrastructure layer (cloud, edge and fog computing), in Sections 2.2 and 2.3 we discuss two crucial middleware components, overlay networks and key-value stores, and in Section 2.4 we discuss service discovery, a central functionality in actual distributed deployments (Figure 2.1).

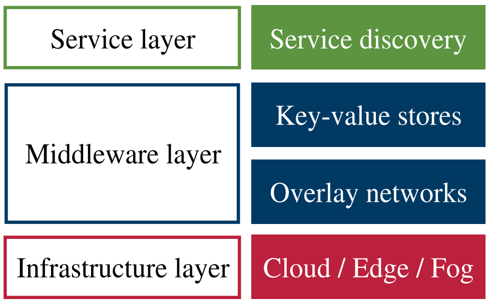


Figure 2.1: A layered view of background and related work.

2.1 Distributed and decentralized utility computing

Utility computing is a provisioning model in which computing resources, infrastructure and services are supplied on demand by a service provider. Its main aim is to benefit from the economies of scale in order to minimize costs while maximizing resource utilization. The most common implementation of utility computing is the Cloud [14].

2.1.1 Cloud

Cloud computing encompasses the infrastructure, software, and policies for providing utility computing to users and companies. Its advances over the last few years have made it the most widely accepted computing environment for most of the IT enterprises worldwide (68%) [105]. Due to its extensive adoption by different actors and complex conceptual model, various definitions of cloud computing exist. However, a widely accepted definition is the one by the National Institute of Standards and Technology (NIST) [116]:

> Cloud computing is a model for enabling ubiquitous, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage,

applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The basic characteristics of cloud computing include: *on-demand self-service, broad network access, resource pooling, elasticity* and *measurability.* On-demand self-service implies that users can obtain or release resources as needed without prompting the provider. The obtained resources can be accessed via well-known network protocols supported by most of the devices. These resources are pooled between different tenants; as a user releases a resource, it becomes immediately available to other users. Elasticity implies that resources for an application can be automatically scaled up or down at runtime depending on the workload. Finally, the cloud uses a metering system for reporting the resource usage, which is leveraged in the pay-per-use business model.

The services provided by the cloud can be categorized in at least three levels: *Infrastructure as a Service (IaaS), Platform as a Service (PaaS)* and *Software as a Service (SaaS).* In IaaS users are provided with the bare physical computational resources (computation, storage, network), while they can run any arbitrary software, including the operating system. Example of IaaS in industry include: Amazon Web Services (AWS) such as Amazon Elastic Compute Cloud (Amazon EC2) [9] and Amazon Simple Storage Service (Amazon S3) [10], Google Compute Engine [101] and Microsoft Azure [38]. In PaaS users are provided with the full software stack which is required to deploy their arbitrary application. Users can configure this software, but do not have control on the underlying infrastructure. Often cloud enterprises which offer IaaS, provide PaaS services too. Examples of PaaS include: AWS Elastic Beanstalk [11], Google App Engine [100], Microsoft Azure, Heroku [72], etc. In SaaS users are provided with access to ready-made applications which are already deployed in the Cloud. They have no access on their deployment and very restricted access on their configuration. Cloud-based applications such as Google Apps/G-Suite [102], Dropbox [49] and Overleaf/Sharelatex [150] are examples of SaaS.

Regarding infrastructure ownership and user accessibility, clouds can be: *private, community, public*, and *hybrid.* Private clouds are owned and managed by a single organization and are accessible only within the organization or by its partners/customers. Community clouds are managed by a group of organizations which share a common interest, and their access is generally available only within this community. Public clouds are often owned by a single organization but they are available to the general public. Finally, hybrid clouds are a combination of any two of the previous options, where they operate as separate entities but communicate through standard protocols.

Most often when talking about the Cloud we refer to public clouds, which are often owned by a single large company such as Amazon or Google. Over the years their infrastructure has evolved from a few datacenters in centralized locations to tens of datacenters scattered around the world. For instance, Amazon currently operates in 64 availability zones (of one or more datacenters) organized in 21 regions in 5 different continents [12]. Despite the fact that cloud companies continuously build new datacenters to increase their presence closer to their customers, this process is not scalable due to infrastructure and economical constraints. In order to further distribute the cloud, new infrastructural models are required.

2.1.2 Distributed Cloud

Internet topology is eminently complex, with various actors managing different parts of it, and constantly changing. This convoluted infrastructure is composed of different subsystems, called Autonomous Systems (AS), which are internally governed by the same policy. AS-es are owned and managed by different organizations, but are interconnected between them, either directly or indirectly. The way these AS-es are interconnected is a subject of research. Two generally accepted and similar models are the Jellyfish [153] and Medusa [27]. According to these models the AS-es are organized in different layers as illustrated in Figure 2.2. Those which have the highest degree of connections are grouped in the top layer, representing the *Core*. As their number of connections decreases, AS-es belong to lower layers. Some AS-es are connected to a single other AS which might belong to any of the layers, these constitute the tentacles of the jellyfish.

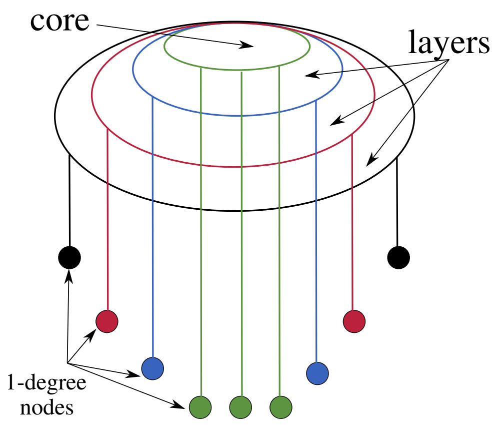


Figure 2.2: The Jellyfish Internet model

Cloud providers manage their own AS which is often at the core layer of this model, while the majority of users are connected in AS-es belonging to the lower layers [89]. For the users to reach the closest cloud datacenter, certain vertical hops in this model are required. That would not be the case if the datacenters were connected in the same AS-es as the users, at the edge of the network.

2.1.2.1 Edge computing

Regardless of the number of datacenters cloud providers build in the core of the global network, the delay to the users connected in low-layer AS-es remains still significant. Such delay can be intolerable for certain latency-sensitive applications, such as augmented reality or connected vehicles. Additionally, other applications are concerned with bandwidth usage, energy efficiency or security issues. Users or devices at the edge of the network create enormous amounts of data that have to be uploaded to the datacenters in the core in order to be processed. Such transfer not only may reduce the performance of the application, but it also overloads the overall network and results in more energy consumed by the datacenters. Moreover, the further from the source data is processes the more complicated it is to restrict its access to unauthorized parties.

Edge computing is a paradigm which aims at providing computing resources and storage as close as possible to the users, in order to address the aforementioned challenges. Defining what constitutes an edge resource and how far it can be from the user remains an open question and subject to various interpretations. This can include from resources in the same Local Area Network (LAN) to small datacenters located one or two hops from the client [172]. In any case, edge computing does not involve user devices and edge nodes are not necessarily connected to the core cloud, but can be interconnected in a peer-to-peer model [57]. Other paradigms such as multi-access edge computing (MEC) [109], cloudlets [147], osmotic computing [163], and fog computing, envision a collaboration between edge and core resources.

2.1.2.2 Fog computing

First introduced by Cisco, fog computing is defined as a highly virtualized platform that enables computing and storage anywhere along the path between the core cloud and the edge resources, in all levels of the network (core, metro and access) [21]. Therefore, fog provides a more generic end-to-end platform than edge as resources are allocated in the core-edge continuum [31]. Fog benefits from the advantages of both cloud and edge, but it comes at the cost of much higher platform complexity. Networking, guaranteeing quality of service, interfacing, monitoring and billing, and security in a heterogeneous infrastructure are some of the challenges of fog computing [170]. Similar to edge computing, there is no consensus whether the fog includes end devices as considered in the context of IoT [20], or the closest datacenter. Figure 2.3 shows a simple conceptual comparison between the three computing paradigms discussed above.

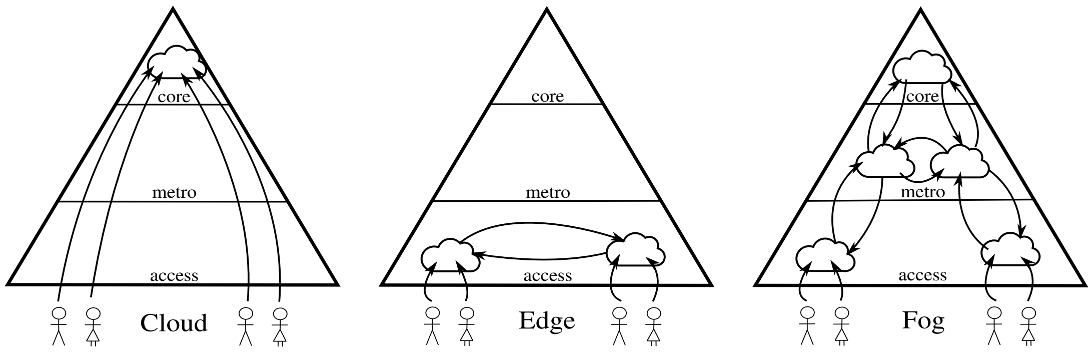


Figure 2.3: Core cloud, edge and fog

2.1.3 Decentralized Cloud

Distributing cloud resources between core and edge often refers to the distribution of computation and data. However, the computation management, metadata and monitoring can still remain centralized even in a distributed cloud. For instance, VM scheduling decisions, metadata about the running VMs, VM image repository, and the real-time status of VM-s can still be handled centrally. While this approach provides various benefits due to its simplicity and complete view of the system, it suffers from similar flaws of centralized clouds.

Centralized management leads to a bottleneck for reliability and availability. In order to overcome this, replication mechanisms are required. These mechanisms introduce significant amounts of traffic to the network. Monitoring information from all the distant edges needs to be centrally collected and displayed on nearly real time. Moreover, as the number of geo-distributed resources or datacenters increases, the centralized management suffers from inherent scalability issues which cannot be easily solved without decentralization.

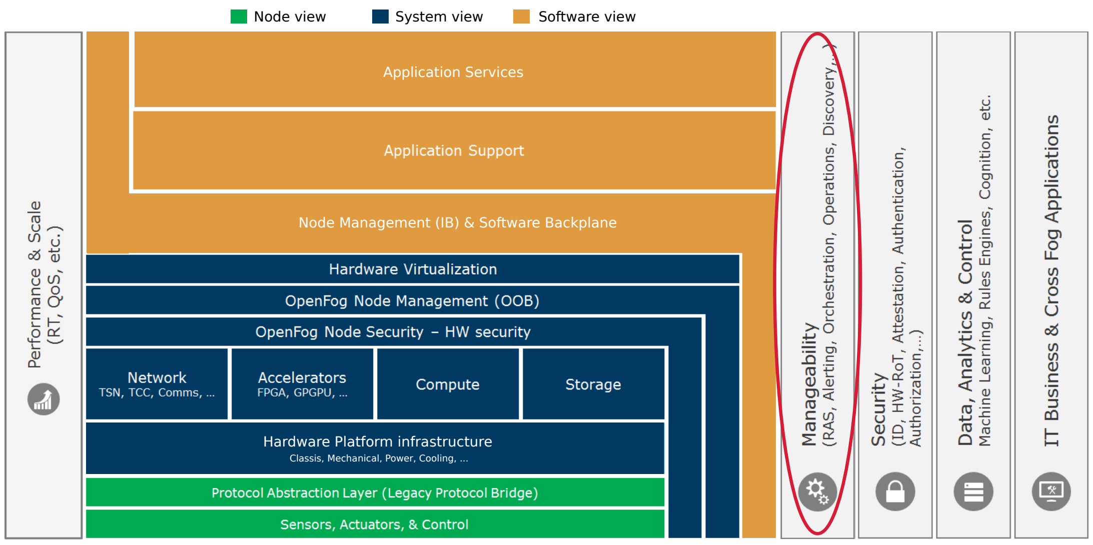
The decentralized cloud approach employees well known P2P techniques to handle management. In [33] the authors argue that a significant portion of cloud services are embarrassingly distributed, meaning that there is little interaction between server pools. This can be reflected in the cloud management, since often metadata, status and images of the VMs starting in a local edge are not required in other edges. In Nebula [30], specific stable nodes (computing and storage master) are chosen to act as managers for other nodes. In [15], authors describe a P2P cloud system based on gossiping for maintaining the overlay, a slicing system for provisioning resources, and a monitoring system for each slice. In [176] authors detail different strategies of how small datacenters collaborate in order to provision requests which can not be handled by a single site.

Recently research and industry are considering the use of blockchain for cloud management in order to improve reliability and traceability. Despite the current enthusiasm around this technology, its current limitations, such as high latency and resource usage, are restrictive to the scope of its applicability [171]. However, various prototypes are starting to emerge. The design of a layered fog architecture based on Software Define Networks (SDN) and blockchain is described and evaluated in [151]. In industry also various initiatives such as Ethernity [51] or iExec [76] are being developed around this idea. Despite the numerous efforts, due to the number and diversity of the involved actors, a standard decentralized fog platform is still missing.

Nevertheless, a general agreement on the nature of the building blocks constituting such platform seems still attainable.

2.1.4 Building blocks

A major effort into defining a standard reference architecture for the fog was done by the OpenFog Consortium [78] and is shown in Figure 2.4. This abstract model distinguishes three views (in different colors) and five perspectives which are cross-cutting concerns involved in various views (in grey). While the lower views, node and system are concerned with hardware and virtualization, the software view comprises the cloud management and the services it provides. We focus further on this view and one the perspectives concerned with this view, manageability.



Node view
System view
Software view
Performance & Scale (RT, QoS, etc.)
Application Services
Application Support
Node Management (IB) & Software Backplane
Hardware Virtualization
OpenFog Node Management (OOB)
OpenFog Node Security – HW security
Network TSN, TCC, Comms, ...
Accelerators FPGA, GPGPU, ...
Compute
Storage
Hardware Platform infrastructure Classis, Mechanical, Power, Cooling, ...
Protocol Abstraction Layer (Legacy Protocol Bridge)
Sensors, Actuators, & Control
Manageability (RAS, Alerting, Orchestration, Operations, Discovery,...)
Security (ID, HW-RoT, Attestation, Authentication, Authorization,...)
Data, Analytics & Control Machine Learning, Rules Engines, Cognition, etc.
IT Business & Cross Fog Applications

Figure 2.4: OpenFog reference architecture (adapted from [36])

The management layer ensures that each resource goes through a life-cycle which includes: commission, provision, operation, recovery and decommission. An important part of management is orchestration. Current fog challenges require the design and development of orchestration mechanisms that can handle tasks such as: scheduling, path computation, discovery and allocation, and interoperability [162]. Enabling all these mechanisms requires a logical network between resources which facilitates decentralized management of these tasks, while complying with the pillars of Fog [36].

The Discovery initiative [17] aims at unifying orchestration and resource management under a distributed operating system based on P2P algorithms which considers locality as the main priority. They advocate for overlays with minimalist design, which ensure high scalability, reliability, and most importantly locality-awareness. They argue that such overlay will provide the basic building block of the platform, on which other higher level overlays and orchestration tasks are built. An overview of potential overlays is given in the next section.

2.2 Overlay networks

Overlay networks are logical networks of nodes and links built on top of an existing network with the purpose to provide an additional service. They are typically deployed on top of the Internet and most commonly used in P2P systems. The way nodes in the overlay are interconnected defines its topology. A topology aims at optimizing the additional service or functionality that the overlay provides. Based on the topology, the overlays can be categorized in two broad classes: structured and unstructured.

2.2.1 Structured overlays

The topology of structured overlays often aims at optimizing point-to-point message routing in a probabilistically bounded small number of hops. This primitive is often the basis for providing other decentralized services such as: network storage, content distribution and application level multi-cast. These overlays are scalable, resilient and provide fair load balancing.

Routing model. Nodes in a structured overlay are often assigned a uniform random identifier, called the node ID, from a large ordered identifier space. Each node is *responsible* for a partition of this identifier space logically close to its ID. The main primitive these overlays provide is *lookup(ID)*, meaning that given an ID, starting from any node, the overlay is able to collaboratively locate the node responsible for that ID by continuously forwarding the message closer to the destination. In order to efficiently route a message each node maintains a routing table, consisting of the IDs of specific nodes and their respective IPs. The IDs of these nodes selected to be in the routing table must comply with a certain distribution in the identifier space, such that at each forwarding step, at least half of the logical distance between the current node ID and the destination is reduced. For this reason often the number of hops in structured overlays is often of the order $O(logN)$. This routing model is the basic greedy approach based on logical distance reduction. Different structured overlays build routing mechanisms on top of this approach.

Maintenance. Overlay networks are dynamic and therefore nodes might join or leave the network at any point in time. These unexpected events are commonly referred to as *churn*. The overlay must maintain its structure in presence of churn in order to provide correctness and efficient routing. This means that each node must continuously verify if their routing table entries are functional and if they still satisfy the distribution criteria mentioned above. This is often done by proactive and periodic protocols which verify the health of the structure and update it properly to reflect any potential change. The more often these protocols are scheduled the lower the probability for failures to have an impact on performance and its stability. However, this service comes at significant traffic generation costs.

Distributed Hash Tables (DHT). DHTs are the most common implementation of structured overlays. The main functionality of a DHT is to evenly distribute data among overlay nodes and be able to efficiently locate it in a decentralized manner. The main component of a DHT is the hash function, which maps any given data of custom size into an identifier of a fixed size. This function can map any data or metadata into the identifier space of the overlay, assigning like this the node responsible for any given data. A common criteria for the selection of such a function is the ability to evenly distribute data in the identifier space. Once the data is distributed among nodes, locating it is done by applying the same *lookup* primitive and routing algorithm as for locating other nodes.

Data resilience. Maintaining the structure in case of churn is one aspect of fault tolerance, another aspect is to avoid losing the data stored on the nodes which leave the network. A common technique for alleviating this problem is to replicate data in other nodes. Often replication schemes involve predecessors or successor nodes in the identifier space, or other nodes which are decided by a convention shared by all the nodes in the network. The responsible node is in charge of keeping the different replicas updated in case of data modifications. Depending on the location of replicas and the frequency of data updates, maintaining replica consistency can come at a high bandwidth cost. Additionally to avoiding data loss, another aspect of resilience is the ability of the overlay to maintain a correct and fair distribution of data in case of churn. Hence, newly joining node should be assigned data corresponding to their responsibility and in case of leaving nodes, their data are distributed among the remaining nodes. The mechanism which recovers data from replicas and redistributes data according to the present nodes, is often triggered by the periodic overlay maintenance mechanism mentioned earlier.

2.2.1.1 Notable structured overlays

In the late '90s - early 2000s given the sudden burst in digital data, there was a shift in interest from centralized client-server architectures to decentralized P2P ones, especially in the file sharing context. This was reflected in a massive body of research on structured overlays which resulted in numerous DHTs being proposed in the upcoming decade. We consider here just of few reference DHTs which ideas were adopted and improved by other later ones.

The classic five. Some of the most notable initial DHTs include: CAN [140], Pastry [145], Kademlia [114], Chord [155], and Tapestry [175]. These are very well-known DHTs and subject to several survey studies where they are compared and confronted [70, 161, 106]. Therefore, we briefly summarize here a few details regarding their common features and their peculiarities. These overlays generally provide probabilistic guarantees that message delivery is done in a logarithmic number of hops, with the exception of CAN for which this number increases faster as the network grows. Similarly, except CAN which considers nodes and data as points in a d-dimensional space, all the others use consistent hashing to generate IDs in the identifier space. All of them use greedy routing which consists on reducing the logical distance to the destination at each hop. Topology-wise Chord and Kademlia have a circular topology, but their distance functions are rather different. Chord uses the clock-wise difference in a circular ID space, while Kademlia uses the *xor* function. On the other hand both Pastry and Tapestry are based on a Plaxton mesh [136] topology where nodes are selected in the routing table based on prefix matching criteria. Finally, all these overlays use heavy mechanisms to deal with churn and their maintenance costs are significantly high due to the relatively strict constraints on routing table entries, especially in Chord, as others allow a certain degree of flexibility. An alternative line of research focused on further relaxing these constraints.

Flexible routing tables. Inspired by small-world networks [85], both Symphony [108] and Viceroy [107] augment their ring-structures with a few random long range links, yet satisfying a specific probability distribution. Symphony uses a particular harmonic distribution, while Viceroy introduces a multi-ring hierarchical structure where long distance rings are those between rings. The benefit in maintenance for Symphony is that links have to be updated only if they do not satisfy the distribution anymore which requires a high degree of churn. For Viceroy, its structure limits the number of maintenance operations to a constant number, but it still remains a complex structure to maintain. Similar to the previous group, the lookup complexity of these two overlays is logarithmic. Later research focused on reducing this complexity to a constant.

O(1) overlays. In certain contexts even a logarithmic routing complexity might be considered expensive, especially if the delay of hops involved is not insignificant. In these cases, overlays can trade an increase of the routing table size for less hops. In Kelips[65] nodes are divided in $k = \sqrt{N}$ affinity groups. Each node has a link to all the nodes in its group and to a node from each of the other groups, resulting like this in $2 * \sqrt{N}$ entries in its routing table and providing a 2-hops routing in the worst case. In ZHT [98] and D1HT [119] the routing table consists of all the nodes present in the network. In this case it is difficult to talk about a structure, but the way data are distributed is still based on a hash function and common replication techniques. These DHTs can be applicable in environments with low churn, but otherwise the maintenance of a synchronized view of the network by all the nodes becomes expensive in terms of bandwidth consumption. An alternative way to reduce the number of hops is to adapt the overlay structure to the application running on top.

Application driven. In addition to the structure which guarantees message delivery from any node to any other node, overlays can be augmented with additional links which favor the traffic patterns of the application built on top of it. Based on the observation that nodes participating in resolving a query are likely to be involved in similar ones in the future, direct links are created between those nodes [87]. In SPROUT [110], social links are added to standard DHTs use the trust relationship between certain nodes in order to improve security. In other cases, instead of new links, replication is used to reflect application behaviour. In Beehive [138], replicas of the requested objects are placed along the path from which the requests for them comes, reducing like this the number of hops.

Structured overlays can be categorized in various ways and a full-fledged taxonomy is not always possible. Often they aim at shortening the route to the object, but in other cases they target high throughput routes, higher security and better load balancing. A large number of studies focus on latency-awareness for which we have dedicated a separate section, but before we briefly describe the unstructured and hybrid overlays.

2.2.2 Unstructured overlays

Differently from structured overlays which select entries in the routing tables based on strict criteria, the unstructured overlays are much more flexible, open and dynamic in terms of routing tables. This absence of structure makes them inefficient for unicasting or point-to-point messaging, but that is not their goal. These overlays specialize in broadcasting or message dissemination. They provide probabilistic guarantees that a message will reach all or a specific group of nodes in the network. The fact that they do not maintain a structure makes these overlays very resilient to churn. However, this comes at the cost of redundant messages and high network traffic. Along with information dissemination these overlays can be used for data aggregation, synchronization, load balancing and, as we will see in the next section, for overlay topology management.

Dissemination model. In these overlays, similar to the structured ones, each node keeps a list of other nodes in its routing table, called the partial view. As mentioned earlier, there are no strict criteria on the logical IDs of the entries in these views. They can be selected completely at random, or, in some cases, based on flexible criteria such as proximity or view size. Message dissemination often follows a network flooding-like model. Whenever a node receives a message, it forwards to all the entries in its partial view. If the receiving node has not seen this message before, it continues propagating it to its entries, until with high probability the message reaches all the nodes in the network. Basic flooding is not scalable and the message overhead is quite significant. A special type of flooding are the gossiping protocols. Gossiping relies on periodic communication rounds in which a node selects a limited number of entries from its partial view, called fanout, and forwards them its information. The fanout number is crucial for the success of the message reaching all the nodes, but it also controls the redundancy of messages. In gossiping information is generally disseminated slower then in basic flooding but it is less implosive in terms of redundancy.

Membership maintenance. Unstructured overlays need also to deal with nodes joining or leaving the network, by updating their partial view accordingly. Fortunately, this does not require additional routines because the algorithm used to disseminate messages can be used also for disseminating information about active nodes. Membership maintenance is done through a more generic mechanism called *peer sampling*, which is typically based on gossiping [81]. At each round, nodes exchange details about their partial views, and they update their views accordingly, by introducing newly joined nodes, or removing inactive ones. Peer sampling does not only update the views with the active nodes but also determines which nodes should be in the partial view based on custom criteria which vary from one overlay to another.

2.2.2.1 Notable unstructured overlays

Unstructured overlays were introduced even earlier than structured ones in the P2P file sharing context. In the early 2000s many applications built on this kind of overlays were very popular.

Early systems. Early versions of systems such as Gnutella [144] and eMule [88] were using query flooding in order to locate files. In later versions they were improved by either using some special peers which would contain more information (superpeers or ultrapeers), or they would introduce some structured overlay for handling file localization. In other cases such as BitTorrent [35], which continues to be a widely used protocol, centralized trackers are used to keep track of the peers sharing a file. Outside the context of file sharing, unstructured overlays are commonly based on gossiping.

Gossip-based. Gossip-based overlays may vary in different aspects such as: the peer sampling strategy, the selection of the entries where the message is forwarded or the way the overlay is maintained. Peer sampling in HyParView [95] attempts to create minimally connected graphs on top of the original overlay in order to minimize the number of messages during gossiping. However, other links are used for membership management in case the minimalistic graph becomes disconnected. In Scamp [56] and CLON [113] authors use a probabilistic model for accepting new entries based on the number a node has currently. This allows to adjust the fanout to the number of nodes in the network. CLON is an adaptation of Scamp for multi-clouds. It improves Scamp's partial view exchange by favoring nodes in the same datacenter more than those in different ones. In terms of maintenance, it can be done either periodically, as in HyParView, or reactively, as in Scamp. Reactive gossiping happens only when nodes detect some change. Overlay maintenance is also treated in a separate section given the importance it has on our work.

Bio-inspired. Gossiping itself is inspired from the spread of viral epidemics, however, some overlays exploit further observations done in the field of biology in order to provide fast and reliable routing/searching and efficient overlay maintenance. Even though these overlays fall under the category of unstructured, their dissemination model is not necessarily gossiping, but it is rather based in swarm intelligence. A typical example of swarm intelligence is the way ants find the shortest path to the food based on the Ant Colony Optimization (ACO) theory [47]. Inspired by this theory, BlatAnt [24] aims at constructing and maintaining an optimal overlay with a small diameter. It determines which nodes should be connected and disconnected based on their distances which are calculated by information given by some ant-like agents which roam the network and distribute information about it. In another study [16], the authors describe a searching mechanism in unstructured overlays based on the humoral immune system. The searched items are considered as antigens, while the query message is the antibodies which are replicated and aim at tracking down the antigens. Similarly, P2PBA [43] uses the foraging behaviour of bees to locate items of interest. Instead of providing the actual nodes holding the searched items, this algorithm provides rather an area where the probability of finding the item is high. This results in a great improvement with respect to message redundancy, since areas with low probability for finding resources are easily discarded.

Unstructured overlays including bio-inspired ones provide highly reliable mechanisms which can be used by other overlays built on top of them as we detail in the next section.

2.2.3 Hybrid overlays

In the previous sections we saw that while unstructured overlays specialize in broadcasting and maintaining shared information, structured ones provide efficient unicasting and resource localization. Therefore, these overlays are rather complementary in the features they target. For this reason significant research has focused on overlays which combine both structured and unstructured principles.

Maintenance model The way these overlays are combined is by letting the unstructured overlay deal with the membership maintenance. This is done by configuring peer sampling to select nodes based on the criteria required by the structured overlay. This technique can be used not only for maintenance, but also to build the structure from scratch, in which case the structure emerges after a few rounds of gossiping. Handling churn can be done either entirely on the unstructured layer, or the two layers can collaborate in order to increase the efficiency of recovering. In the latter case, the structured overlay can be notified by the underlying layer of the changes and can help to locate the nodes needed to rebuild the topology more efficiently.

2.2.3.1 Notable hybrid overlays

Often hybrid overlays have tightly coupled unstructured and structured layers which are configured to achieve a specific goal. For example, in some cases the lower (unstructured) layer specializes in providing the upper layer with proximity low-latency nodes, such as in Fluidify [142] which we revisit in the next section. In other cases, such as Chams [8] gossiping takes into account the reliability of a node so that the structure can be build mainly on reliable nodes. In Kelips, which was described earlier, gossiping is used to maintain the list of members in the same affinity group.

Collaboration between tightly coupled overlays is optimal, but it limits their independent usability. In order to overcome this problem, T-Man [80] introduced a generic overlay building mechanism which provides configurable gossiping. T-Man provides an interface to higher layers where they specify the kind of peer sampling by specifying a ranking function. As a result, various hybrid overlays have been built on top of T-man. For example, T-Chord [123] uses T-Man to bootstrap and maintain Chord without the need of additional maintenance protocols. Similarly, Vitis [137] uses T-man to create small-world networks based on nodes interested in similar topics in the context of a publish-subscribe mechanism.

Gossiping can be used by structured overlays also for other purposes not strictly related to the overlay maintenance. In P-Grid [3], gossiping is used to update different replicas of an object in case of a new version. GridVine [2], built on top of P-Grid utilizes semantic gossiping for providing semantic interoperability [40]. In this way gossiping can be used join or associate different definitions of the same object into a single key which is later used in the structured overlay search.

Bio-inspired overlays can also be used to construct or optimize structured overlays. An example of this is Self-Chord [53], where swarm intelligence inspired by ant colonies is used to improve maintenance and load balancing in Chord. In Self-Chord there is no strict relation between nodes keys and resource keys, and therefore ant-like agents can move similar resources to the same nodes according to a custom criterion called *peer centroid.* This enables range queries and provides lower maintenance in case of churn. The same concept can be used for other reference DHTs, such as Self-CAN [61] and others.

After having discussed the three main kinds of P2P overlays: structured, unstructured and hybrid, we focus on how these overlays implement two of the most relevant features in this study, namely locality-awareness and reducing maintenance overhead.

2.2.4 Locality-awareness

Early structured overlays focused mainly in finding the best tradeoff between the number of hops to the destination and the size of the routing table. However, given that the logical distance does not reflect the physical one, getting closer to the destination logically might result in long expensive hops back and forth in the physical network. In a WAN implementation of these overlays, such costs are added up and result in significant delays. For this reason, locality-awareness is one of the most targeted features in structured overlays, but to some extent also in unstructured ones.

2.2.4.1 Structured overlays

The ratio between the latencies of a logical path and the physical path from the same source to the same destination is called the *stretch*. The more a logical path diverges from the physical one, the higher the stretch is. In structured overlays, locality-awareness aims at minimizing the network stretch. According to Castro et al. [29], locality-awareness can be introduced into structured overlays through different mechanisms, namely: locality-aware neighbor selection, proximity routing, and geographic layout.

Locality-aware neighbor selection. It consists in choosing as entries in the routing table the physically closest node among viable candidates. This technique is used in some of the classic overlays such as Kademlia, Pastry and Tapestry, which allow some degree of flexibility in the neighbor selection. Also T-Chord relaxes Chord's constraints for having a link at a precise logical distance, by allowing a range of values around this distance, so that the ones with the lowest latency are selected. In the case of T-Chord low-latency alternatives are provided by the underlying gossiping layer. A similar approach is used also in DHTs based on small-world networks such as Symphony where the degree of flexibility is even higher.

Proximity routing. It consists in taking routing decisions not only according to the remaining logical distance to the destination, but also on the delay of the next hop. It is a lighter approach since it does not require any routing table modification and it is more reactive to latency changes. This concept was initially introduced in CAN, but it is adopted also by later DHTs such as Hypeer [148]. Hypeer aims at introducing latency-awareness in Chord by changing the order of its hops, in such a way that initial long logical hops can be traded for shorter but low latency hops which also reduce the logical distance. However, the approach followed by Hypeer requires a uniform distribution of IDs over the identifier space, which makes the ID assignment for new nodes more complex.

Geographic layout. It consists in assigning node IDs in such a way that logical distances between nodes reflect also the physical ones. This can be done implicitly, where logical IDs are updated to reflect newly discovered low-latency nodes, or explicitly, where IDs are assigned based on prior knowledge of the latency between nodes. Fluidify [142] follows the implicit approach. Additionally to the logical neighbors, a node keeps a list of physically close neighbors and periodically it evaluates if exchanging the IDs between a physical and logical neighbor results in a benefit for the overall system. This approach might lead to expensive periodic data relocation.

In the explicit approach often the ID of a node reflects its location. In eQuus [103] a node discovers in the joining phase a group of other physically close nodes called a *clique*, which share the same ID. Nodes within a clique know about each other and share the same data. This provides locality-awareness within a clique but not between cliques. A similar idea is also used in LHDTs [168], where the autonomous system number is embedded in the node identifier in order to distinguish between local and global nodes. Brocade [174] and Expressway [169] introduce an additional overlay of carefully selected super-peers (landmarks), representing node clusters (often AS-es), which provide physical insights into the logical routing.

In some other cases more than two locality levels can be devised by using hierarchical overlays which result in more structured IDs. For instance in [117] independent ring-overlays are connected through gateway nodes forming a higher level ring. A multi-ring approach is also used by Skipnet [69], but the path locality is rather guaranteed by distinguishing between the numeric IDs and name IDs, where the latter are assigned in such a way that physically close nodes are assigned the same prefix.

Building locality-aware overlays requires nodes to be provided with a latency measurement, often the Round Trip Time (RTT), to other nodes, which might not even be in their routing table. This is usually done either by some ad-hoc discovery mechanism based on active probing of random nodes, or provided by an underlying gossiping mechanism. However, these mechanisms either do not scale well or are relatively expensive in terms of traffic. A lighter alternative is to use network coordinates.

Network coordinates. The idea behind network coordinates is to treat each node as a point in a d-dimension euclidean space with specific coordinates. These euclidean distance between two points should reflect the latency between the respective nodes. In this way, just by knowing the coordinates of a node, one can predict the latency to that node without the need of contacting it. There are a couple of ways these coordinates can be computed. Early approaches, such as Global Network Positioning (GNP) [129] used landmarks which are nodes whose coordinates are well-known so that the coordinates of the newly joined nodes were calculated using triangulation. However, this approach has some disadvantages regarding scalability and security as landmarks are a bottleneck or might not even be available. Later approaches extended the landmark-based one by allowing every node in the system act as a landmark. The most notable algorithm in this category is Vivaldi [41]. Vivaldi is based on the metaphor of a physical spring system where nodes are unitary masses connected to each other through springs. The rest length of the spring between two nodes represents the measured RTT, while the actual spring length represents the estimated RTT between them. Since springs tend to adjust their actual length to match the rest length, the more nodes interact the more nodes will push or attract each other by adjusting their coordinates until the system stabilizes. This approach is fully decentralized and any interaction between nodes is used to update the coordinates, differently from the landmark-based approaches. This means that potentially application traffic can be used to calculate these coordinates in a lazy way.

2.2.4.2 Unstructured overlays

As mentioned earlier, unstructured overlays are often used to provide locality-awareness as a service to structured layers built on top of them. Since gossiping provides interaction between random peers, it can be used to discover local peers by configuring low-latency as a criterion for selecting nodes in the peer sampling process. This is the case in many hybrid overlays including the aforementioned Fluidify or T-Man. In the later one, this is done by specifying a simple ranking function based on latency.

In addition to providing it as a service, locality-awareness is also useful within the unstructured overlays themselves. Its implementation mainly consists in prioritizing low-latency entries in the partial views rather than high-latency ones. The motivation for this depends on the specific application. In the context of information dissemination, oftentimes the value of information is higher within a range of nodes where it is generated than in distant ones, which might very well receive it later. In other cases, favoring local nodes minimizes redundant long distance links by saving some bandwidth, without affecting the performance of the application.

An example of locality-aware gossiping is CLON which was mentioned above. CLON considers a multi-datacenter environment where information needs to be disseminated with a higher priority within one datacenter before moving to the next one. Prioritizing local nodes is done in both levels, in peer sampling and message dissemination. A similar concept but which can be applied to any unstructured overlay is the Localizer [112] algorithm. Localizer uses a probabilistic model to rearrange the links produced by a standard unstructured overlay so that the final topology resembles more to the physical network.

2.2.5 Reducing maintenance overhead

When describing structured overlays we remarked that the maintenance of their structure was done through specific protocols, or in case of hybrid overlays, this service was provided by the underlying gossiping mechanisms. Early DHTs promoted the idea that in order to provide efficient routing certain invariants regarding the structure must be fulfilled. As a result their maintenance protocols were generally proactive, involving periodic probing of routing table links and immediate correction in case of staleness or appearance of new nodes. However, in large systems, possibly where multiple overlays operate, the overhead of such maintenance becomes significant. Therefore several studies focused on reducing the maintenance overhead of structured overlays.

MSPastry [28] uses the structure itself to reduce the number of sent probes. This can be done either by using events, such as node joining, to disseminate additional maintenance information, or by partitioning the failure detection responsibility so that only specific nodes are notified when a failure is detected. They show that with this technique they can achieve robustness at high rates of churn with overhead comparable to the one unstructured overlays.

Along the same lines, Chord^2 [82] argues that a hierarchical two-layer ring structure based on super-peers reduces significantly the maintenance cost. This is under the assumption that the failing nodes are mainly in the low-level rings which are small and require less maintenance, while the super-peers which participate in the larger rings are more stable.

In other studies reducing maintenance lies in the flexibility of the entries in the routing table. Similar to Symphony, inspired by Kleinberg small-world networks, the authors of [84] present a stochastic maintenance strategy that reduces the overhead to the minimum required to avoid partitioning. For each node they define a single parameter which defines the quality of the current distribution of long links. Maintenance actions are needed only when this parameter goes beyond a certain range.

The conditions which trigger maintenance traffic were also the focus of [1]. This study describes two reactive alternative to periodic protocols, namely correction on use (CoU) and correction on failure (CoF). While the CoU corrects stale links as soon as detected, the CoF is even lazier and corrects them only when alternative paths are impossible. The study compares the two mechanisms and the proactive approach on different levels of churn and query frequencies and concludes that they are more efficient in most of the combinations.

The problem with reactive maintenance is that it generates even more traffic at the moment where failures have occurred, which increases the probability for more failures. This is studied in Bamboo [143], which a provides a congestion-aware recovery periodic protocol. Even though periodic, Bamboo schedules the maintenance only when the state of the overlay is not under stress.

Fuzzynet [62] follows a more extreme approach by dropping any explicit connectivity and data maintenance requirement. The authors argue that maintenance not only introduces overhead, but also does not guarantee that the structure is intact. Due to network configurations certain nodes are unreachable from specific other nodes which violates structural invariants. Fuzzynet relies only on the actions when new peers join the network and on alternative routes. They show that with sufficient amount of neighbors, data can be retrieved with high probability even under high churn.

Reducing message overhead is also targeted by unstructured overlays. The proactive and reactive approaches apply also in membership maintenance based on gossiping. As mentioned earlier, Scamp is one of the gossiping protocols in which partial views are updated only as a result of an overlay event, such as nodes joining. However, this lazy maintenance can result in some nodes being isolated, since nodes which had them in their partial views might have been disconnected. In case a node has not received a message for a given time, it will rejoin the network.

Other overlays such as HyParView or protocols like Localizer tackle this problem form the structural point of view. Hyparview creates minimally connected graphs while it keeps a passive partial view just in case partitions are created. Localizer on the other hand reorganizes links in such a way that redundant links which do not match the underlying network are removed. In both cases created graph is more lightweight and requires less messages for being maintained.

We now move to a related topic which shares many characteristics with both structured and unstructured overlays, but addresses additional challenges of modern cloud applications, the key-value stores.

2.3 Key-value stores

Originating from a file sharing context, overlay networks, specifically the structured ones, could be seen as a first step towards distributed databases. They were offering some basic database primitives such as key-based, range or even multi-attribute queries, but the data they were dealing with were often read-only. Nevertheless, providing a full-fledged distributed database was not in their scope.

For many years the database sector has been dominated by the *relational model* and the Structured Query Language (SQL) which provided a very versatile tool to retrieve and combine data, but which relied on a centralized architecture. In the early 2010s, with the advent of cloud computing many applications were ported from a centralized setting to a distributed one, but the relational model failed to adapt to this new environment. Its strict dependency between tables makes it inflexible and difficult to scale horizontally. As response to this, alternative non relational, NoSQL models were proposed, including here key-value stores.

In key-value stores data is modeled according to the way it is accessed by the users, so that a request involves possibly only one entry in the database, which is identified by a *key* and has all the required data as a *value*. Compared to the relational model, this is much simpler, it provides an easier integration with object-oriented languages and most importantly, it scales very well horizontally. Key-value entries can be replicated in different database instances without the need to copy a whole chain of related tables. To a very large extent this converges with the idea of DHTs, even though it originates from the database perspective. The main difference is that data in this case is not read-only and replication requires a consistency model.

CAP theorem. In a distributed data stores, such as key-values, there are three features central to their design, namely consistency, availability and partition tolerance. In order to provide consistency a data store must guarantee that every read operation on a key returns the most recent written value or an error. A data store provides availability when every request receives a non-error response, which might not necessarily be applied to or return the most recent value. Finally partition tolerance is provided when the data store continues to be functional despite the failures or delays of the network between nodes.

The CAP theorem [23] states that it is not possible for a distributed data store to provide more than two of these features simultaneously. In other words, a data store can be tolerant to network partitions only if it trades consistency for availability, or vice versa. This yields three possible combinations of these features: consistent-available (CA), consistent-partition tolerant (CP), or available-partition tolerant (AP) (Figure 2.5). Given that dealing with network failures is critical in a distributed environment the CA combination is not very applicable. Indeed, relational data stores fall in the CA category, and therefore their adoption in the cloud

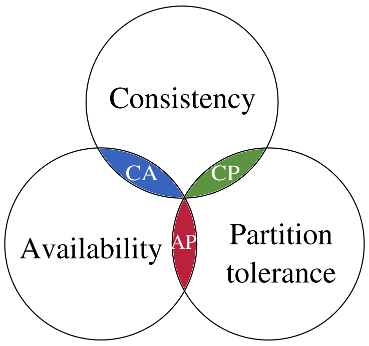


Figure 2.5: The CAP theorem

environment is difficult. NoSQL data stores instead, are split between CP and AP depending on the particularities of the application. However, more often data stores tend to provide the so called BASE properties.

The BASE model. In contrast to the relational data stores ACID model, which promotes consistency over availability, most modern NoSQL data stores implement the BASE model, which stands for Basic Availability, Soft-state and Eventual consistency. The main idea of this model is to increase availability by downgrading from strong consistency to eventual consistency. This means that the system can afford temporary inconsistencies, but given a time of no updates it will converge to a consistent state. The soft-state property is related to the fact that data might have a time to live after which, if not refreshed, it goes away. However, this property is not very common in current NoSQL data stores.

Other types of NoSQL. In addition to key-value stores, NoSQL data stores can adopt various other data models. We consider here a few of them which are more common, such as column, document and graph data stores. Column data stores are similar to the relational databases in the sense that information is organized in rows and columns. However, their data access pattern is specialized to maximize the performance of queries which require data from a few columns but many rows in the same table. Typically each value is associated with the row identifier which enables the linking of different columns, resembling a reverse key-value store (multi-value-key store). Document data stores are optimized to serve queries on isolated objects, called documents, which do not comply with a specific schema. Documents can conceptually belong to the same class (collection) but do not have to share the same properties. They are similar to standard key-value stores, where values are rather modeled in a JSON or XML format. Finally, graph databases focus on optimizing queries which involve complex relations between entities. These data stores make use of adjacency or incidence matrices to represent these relations. They are quite more specialized then key-value, column and document stores which are relatively similar.

2.3.1 Notable NoSQL data stores

The NoSQL movement was mainly driven by the big cloud companies which were the first ones to witness the limitations of the relational model in their products. In order to address some scalability issues occurred during the holiday season in 2004, Amazon introduced Dynamo [42], a highly available key-value store. Dynamo is based on a collection of existing concepts, some of which we have mentioned earlier. It is an eventually consistent, AP data store which is always writable, regardless of the possible different data versions. Conflicts are resolved on reads, using syntactic or semantic reconciliation based on vector-clocks. It maintains a ring topology where data is distributed using consistent hashing and all nodes maintain a complete view (zero-hop DHT) of the membership, which is handled by a gossiping protocol. Data are replicated in the ring and a minimal quorum-based protocol is used to validate read and write operations. In order to deal with inconsistencies between replicas in case of failures, Dynamo implements a synchronization protocol which is based on hash trees, which efficiently detect data inconsistencies by comparing root hash values.

The Dynamo paper inspired other similar products by other well-known internet companies, such as Cassandra [91] by Facebook, and Voldemort [156] from Linkedin. While Voldemort is an adaptation of the Dynamo concepts for their particular use case, Cassandra provides some additional features. Cassandra introduces a column-oriented semi-structured data model, which provide column clustering and ordering in order to facilitate certain types of queries. These queries can be defined using an SQL-like language called Cassandra Query Language (CQL).

While data stores based on Dynamo fall in the AP category, a few other NoSQL stores trade availability for consistency, hence they fall in the CP category. One of the most popular key-value stores in this category is Redis [141]. One of the particularities of Redis is that it is an in-memory data store, meaning that data is kept in the main memory for fast access, making it very useful for real-time applications. However, disk snapshots are possible too. When deployed in a cluster, Redis uses range or hash partitioning (sharding) to distribute keys among nodes. Data are replicated, but a single source of truth is used. This means that a master node is responsible for read/write operations and keeping the slave replicas updated. Only when a master fails, one of the replicas is elected to become the new master.

Another well-known data store of the same CP category is MongoDB [118]. MongoDB shares many of the properties of Redis when it comes to master-slave replica management and sharding mechanisms. However, it is a JSON-based document store and saves data to the disk. MongoDB provides a JavaScript-like query language which allows key-based, range or even regular expression searches. Additionally, it also provides data aggregation using the map-reduce model present in JavaScript.

Finally, one of the most popular data stores in the category of graph-based is Neo4J [125]. In order to maintain the consistency in a cluster Neo4J nodes are split between core and replica clusters. Core clusters have a leader and are responsible for write operations, which are validated by a majority consensus commit. They eventually update replica clusters which are responsible for reads. However, Neo4J improves on eventual consistency by providing causal consistency. This is done by means of bookmarks which suggest to the replica nodes the state they should have before answering a request. Querying this database is done through a declarative query language called Cypher.

Now that we visited the infrastructure and the middleware layers, we move on to the service layer. In this layer, the tools and concepts discussed above are put into use to serve a functionality which is very important in the way modern applications are designed and deployed, the service discovery.

2.4 Service discovery

From the early days of network accessed resources, such as printers in a LAN, to the current day technologies, such as Internet of Things (IoT), service discovery has been a fundamental functionality in distributed systems. While initially it was about locating hardware devices in a small network, with protocols such as the Universal Plug and Play (UPnP), it slowly evolved to a more generic concept of services and operating in much larger scale networks. These services can be both, hardware and software. Nowadays, hardware services are often linked with resource provisioning, as in the case of clouds, or location of smart devices, as in the case of IoT. However, here we focus mostly on the software services which are the core concept of Software Oriented Architectures (SOA).

2.4.1 Service Oriented Architecture (SOA)

Service Oriented Architecture is a paradigm in which software is organized in "services", which are well-defined modules delivering a specific business activity [134]. Services are self-contained and function as a black box for the consumer, with the constraint that their input and output should be described in a language that is shared by all other services. However, internally they can be heterogeneous and use different technologies and languages.

As opposed to the monolithic architecture, SOA allows reusing the business logic of one service in many different applications, which facilitates the creation of more complex solutions with lower investment. An additional advantage is that SOA removes the need for integration since services are built to be intrinsically interoperable regardless of their provider. To achieve efficient interoperability all services adhere to standard web interfaces and contracts. A common language used to describe services and their interactions is the Web Service Definition Language (WSDL), and a typical messaging protocol used for their interactions is the Simple Object Access Protocol (SOAP).

Service discovery in SOA. SOA is governed by three main actors: the service provider, the service broker (or service registry), and the service consumer. The service provider publishes to the broker its service relevant metadata, including the description, Cost of Service (CoS), and Quality of Service (QoS) it provides. The broker functions as the repository for the services' metadata and location. The interaction with the registry is also based on a standard web specification called Universal Description Discovery and Integration (UDDI) in order to maintain interoperability. Finally, the service consumer contacts the broker in order to find the service which best satisfies its requirement based on their description.

However, despite numerous efforts to distribute and automatize service discovery [139], in practice companies hosted their own centralized repositories and discovery was quite static and manual. SOA was a promising approach promoting a new business model for building software based on standards and cooperation. Unfortunately, in practice SOA specifications varied between implementations and the need for interoperability resulted to be detrimental for its success and growth. A new generation of SOA addressed some of these problems.

2.4.2 Microservices

Although the general idea of interoperability in SOA did not take off, many of its principles still remain very attractive for today's industry demands. The collection of these selected principles constitutes the recently emerged microservices architecture. Microservices are not conceptually different from services in SOA, but might be more limited in their size and scope depending on the required level of granularity. The motivation for granularity in modern applications comes mainly from the facility it provides in code management, deployment, and scalability [128]. Microservices allow to easily assign well-defined responsibilities to different teams within the same enterprise. They enable the partial upgrade of an application without redeploying it entirely. Additionally, they enable to scale up or down a specific part of the application depending on the request load of the user, which would not be possible in a monolithic application.

Differently from SOA services, the scope of interoperability in microservices is between teams of the same enterprise, rather than between enterprises. This eliminates the trust issues in SOA but also the need to use a complex language to describe services and a heavy protocol to handle their interactions [48]. Therefore, microservices commonly provide lightweight Representational State Transfer (REST) APIs, which are based on JSON messages over HTTP. However, recently a new standard for inter-microservice communication is emerging, namely the gRPC Remote Procedure Call [64]. gRPC is based on HTTP/2 and offers request multiplexing and efficient binary data representation. Additionally, it supports bidirectional streaming and automatic load balancing, which require additional implementation efforts in REST APIs. However, gRPC is less adaptable to API changes and it is harder to debug. In this thesis we focus in REST-based microservices as they are still the most commonly diffused ones.

Microservice discovery. One of the particularities of microservice architectures is their dynamism. Encapsulated in lightweight containers, microservices are likely to start, replicate, migrate and stop quite frequently. Service discovery in this case needs to keep up with this constantly changing environment. The currently available service instances and their locations are stored in service registry, similar to the one in SOA, but rather more distributed and more dynamic. Service registration and deregistration can be done either explicitly by the service itself, or it can be provided by a third-party tool which monitors the hosting platform for container status changes.

While in SOA discovery consisted in finding the best fitting service among different providers, in microservices it consists in locating the most appropriate service instance among a set of replicas. What defines the most appropriate instance varies in different contexts. Classic examples include: QoS and CoS considerations, attribute matching, load balancing, etc. A typical example of QoS criterion is the latency to the client. CoS is relevant in a cloud environment where deployment costs can vary between instances. Attribute matching is useful when different versions of the same service exist under the same name (ex. different API versions). Selecting instances based on load balancing criteria is also very critical for guaranteeing the QoS.

There are two main patterns for selecting an instance: client-side and server-side discovery [121]. In client-side discovery, the client queries directly the registry for retrieving all the possible instances, which it then filters according to its criteria. In server-side discovery the client contacts a load-balancer, which in turn queries the registry and then based on the client query and the current load of the services chooses the right instance. While the former method removes an extra step and possibly a bottleneck, the later provides a more aggregated view of the request load and can work with registry agnostic clients.

Monitoring and health checking. Ideally service registries keep an updated view of active services, by excluding those which are down or unavailable. While in the external container level this can be achieved by monitoring the virtualization platform, checking the internal status of a microservice requires additional mechanisms. Currently, there is no standard way for attaining this, but in practice most microservices provide an open health checking API, which can be continuously invoked by the discovery mechanism in a "heart beat" fashion. Whether this mechanism needs to be proactive or can be handled in a lazy way depends on the ability of the application to cope with temporal unavailability.

2.4.3 Notable microservice discovery mechanisms

Microservice discovery mechanisms can be seen as overlay networks or key-value stores in action. Virtually every mechanism described in the previous two sections of this chapter can be the basis for microservice discovery. Microservices metadata are represented as a key-value pair, where the key is the unique identifier of the service (its qualified name) and the value contains information about its location or health status. What discovery mechanisms add on top of key-value stores often includes request proxying, load-balancing and health checking. Given that microservices are a current industry trend we focus on mechanisms used by some of the main companies adopting them.

Although rather generic in its scope a commonly used back-end store for microservice discovery is Zookeeper [74]. Zookeeper provides a sequentially consistent key-value store which has a hierarchical file-system-like data model. Data are organized in a tree where each node has a path and is called a zNode. They are replicated among several servers, one of which is the leader while others are followers, and a majority quorum is required on write operations. One particularity of Zookeeper found to be useful in service discovery are the ephemeral zNodes, which are nodes which disappear when the session of the client which created them is closed. This is useful in a client-discovery pattern, as used by Twitter, because it keeps track of available services without any additional effort. However, in case of server-discovery pattern additional tools are required as in the case of SmartStack [7].

SmartStack [7] is the service discovery mechanism of Airbnb, which is similar to the one used in Twitter [67], it is based on Zookeeper. It has two components called Nerve and Synapse. Nerve is responsible for creating ephemeral zNodes and monitoring the health of services, such that it closes the session in case services become unavailable. Synapse, based on HAProxy [68] acts as a proxy and load balancer between services. Synapse subscribes to events in Zookeeper, such as creation or deletions of keys (service instances) and updates the HAProxy accordingly.

Another general-purpose key-value store commonly used for service discovery is Etcd [37]. It is quite similar to Zookeeper in that consistency relies on a leader which is elected using the Raft [131] protocol, and write operations require quorums. Etcd is used as back-end for service discovery in Kubernetes [55]. SkyDNS [149] uses Etcd too but it provides a basic DNS interface on top of it. Although reliable and thoroughly tested in industrial environments, Zookeeper and Etcd are rather centralized and do not scale well in multi-datacenter settings.

This problem is partially addressed by Eureka [126], by Netflix. Eureka relaxes the consistency constraints of Etcd and Zookeeper which improves its scalability. Similar to Neo4J, it consists of read and write clusters. Read clusters are effectively a cache layer and can be scaled up or down according to the request load. Eureka is a client-side service discovery, meaning that services include a client library which can handle the interaction with the server clusters as well as deal with load balancing. However, it can work with agnostic clients too, by means of an additional tier like Ribbon [127].

A discovery system more adapted to scale in multi-datacenter environment is provided by Consul [71]. Consul is a combination of unstructured overlay concepts and key-value stores. Its architecture is datacenter-aware, and each datacenter consists of client and server nodes. Similarly to Eureka, clients are cache-like nodes which can easily scale, while servers are the one which handle consistency by electing a leader and replicating data between them. Membership within a datacenter is handled using LAN gossiping between all nodes, while datacenter discovery is handled with WAN gossiping between the leader servers of each database. In general data are not replicated between datacenters, therefore a server receiving a request for another datacenter simply forwards it to a node in that datacenter. However, partial replication and caching between datacenters are possible too.

From a more academic perspective, the authors of [22] target a future evolution of service-based applications by addressing the scalability in mega-scale systems. They propose a hierarchical layered distributed registry which takes locality into account. Services are stored locally in the lower layers where their clients are connected. The more a service is used by geographically distant users, the higher in the hierarchy it is stored. However, layers are synchronized horizontally and it is not clear how they are geographically distributed and what the cost of layer synchronization is, especially for lower layers. Additionally, registries and clients are closely coupled and this requires a mechanism to globally identify clients, which does not apply to many real-life scenarios.

Although microservices have an industrial background further research is needed to capture the design requirements of service discovery mechanisms in extremely dynamic and heterogeneous environments such as the fog [59].

2.5 Conclusion

In this chapter we reviewed some of the concepts and the respective literature which serves as background to our contribution. We did so by considering three different layers: physical, middleware and service. In the physical layer we examined the environment our work operates, focusing on decentralized fog and its management. In the middleware layer we revisited the main notions and approaches used to build overlay networks and key-value stores which are strictly related to our main contribution in this layer. A particular attention was aimed at structured overlays, their maintenance and how they implement locality-awareness. Finally in the service layer, we went over the functionality our contribution delivers, which is service discovery in microservice-based applications. In the next chapter we provide a high-level overview and the design principles of our contribution in the middleware layer.

Chapter 3

Designing Overlay Networks for Decentralized Clouds.

3.1 Introduction

Recent technological developments in IoT, data analytics and latency-sensitive applications have pointed out the limitations of the centralized cloud model [124]. Unprecedented volumes of data are generated at the edge of the network and need to be instantly pre-processed in order to propagate only aggregated results for further processing in higher layers of the network [133]. Additionally, latency-sensitive applications, such as augmented reality, demand physically close processing resources in order to be usable. However, current centralized clouds fail to meet these demands. Data is directly transferred to a central processing entity which is often distant from their source. This results in both high bandwidth usage and increased latency.

In order to overcome these issues, increasingly more research has been devoted to alternative architectures such as edge or fog computing. These architectures aim at providing distributed computing resources at different levels of the network, including the edge, where data is generated. However, to this day there is no standard utility computing platform based on such a distributed architecture. For this platform to emerge, it is essential to capture the right design requirements for an implementation that can fully take advantage of this architecture.

An important effort in this respect is done in the Discovery project [17]. This project considers adding some computing resources in each Point of Presence (PoP) of existing backbone networks, by transforming them into small datacenters, as shown in Figure 3.1. Discovery's authors argue that just by having resources close to the users does not fully leverage locality if the management of these resources is still centralized. Therefore, they propose to decentralize the cloud management too.

In contrast to traditional cloud management, in a decentralized setting there is no single controller node having an overall view of all compute nodes. Each node may act as a controller or as a compute node, and has only a partial view of the network. In order to provide management tasks, such as service localization or workflow management, these nodes need to collaborate and coordinate their actions. This can be done by means of an overlay network.

This overlay needs to comply with the principles of fog architectures. The fog is intrinsically minimalistic. Computation needs to be handled no further than where it is feasible and data is transferred no further than where it is required. In the same way, the overlay network should not send messages further then needed and should not generate unnecessary traffic.

As we saw in the previous chapter, traditional overlays originated from a rather generic context, in which the challenges of the fog environment were not necessarily present. Likewise, certain issues they were addressing are not considered pertinent in the current situation. This does not mean that the techniques implemented in these overlays are not relevant in a fog architecture. On the contrary, individual overlays have explored approaches that are admissible

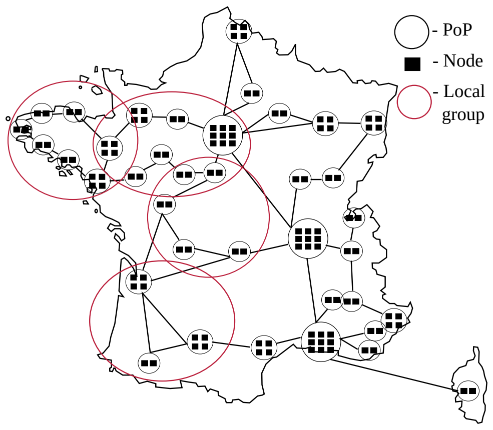


Figure 3.1: Envisioned Discovery infrastructure based on Renater[1].

in similar circumstances. Our goal is to identify and consider their adaptation to a fog context.

In this chapter we analyze the challenges of the fog environment and discuss about the way overlays can address them. We revisit a collection of overlay networks concepts described in the previous chapter which can be applied or adapted to this environment. The goal is to present the general motivation and design choices for our overlay network which we are going to examine in details in the next chapter.

3.2 Fog management

Differently from the centralized cloud, the fog requires computing resources to be distributed in different levels of the network, from the core to the edge. Considering that the vast majority of the fog use cases consist of locality based applications, less and less data make it to the core. This results in a redistribution of user traffic from the links connecting users with the core, to the lower level links connecting users with each other.

However, the distribution of computation has an adverse effect on the platform management if this remains centralized. Computation is delivered by means of Virtual Environments (VE), often Virtual Machines (VMs) or containers, which need to be provisioned, scheduled and monitored. While in centralized settings these VEs tend to have better specifications to accommodate more users (vertical scaling), in distributed settings they have less resources but are more in number and more difficult to reach from a central server. Monitoring such a system results again in a traffic increase at the higher network levels, and can become a bottleneck as the system scales up. In other words, decentralizing computation without decentralizing the management reduces user traffic but increases management traffic.

As mentioned earlier, the premise of the fog is based on locality — the more local the information the more valuable it is. Therefore, central monitoring not only introduces traffic overhead, but it is not very useful since most commonly cloud users would not be interested in the state of the entire system, but rather a small subset of it running in their vicinity. In order to keep the information where it is needed management needs to be decentralized.

Decentralized management requires a structured overlay network in order to provide resource localization. This overlay needs to be very efficient for managing local resources, but also needs to support the localization of distant ones. This can be done using a hierarchical or a fully-decentralized flat structure. Even though hierarchical approaches look more straight-forward, they are difficult to maintain and require complex operations in case of failure. Additionally, mapping a relevant tree architecture on top of backbone (as the one in Figure 3.1) networks is not meaningful, and it is usually done statically [93].

[1]The French research backbone network: https://www.renater.fr

On the other hand, flat approaches are easier to maintain, more resilient to failures and have been proven to scale well in large distributed systems such as file sharing. Yet, maintaining a flat structure may still suffer from the infrastructure's intrinsic centralization — communicating logically between two edges might mean going physically through the core, and hence, not redistribute the traffic. One way to address this problem is to reduce maintenance to the bare minimum by being reactive to compulsory events, rather than proactively generating maintenance events.

In addition to traffic generation, mapping the physical fog structure to a flat overlay is not an easy task either. While overlays do support localization in general, their ability for efficient local resource management needs to be improved. As discussed in Discovery [17], a step towards this is to be able at least to distinguish between *close* and *far* based on a latency threshold which groups nodes in local groups (Figure 3.1). Moreover, even when localizing distant nodes, locality-aware routing is critical to keep traffic only where required. Discovering local nodes and providing locality-aware routing become more challenging when trying to minimize overlay maintenance, but techniques which combine them can be devised.

In order to meet stringent latency requirements, an overlay needs to minimize the number of hops between a source and destination too. Traditional overlays use very limited routing tables, but this limitation is not necessary and can result in a high number of hops. On the other hand, zero-hop DHTs use full views of the system, but do not scale well and generate extra maintenance traffic. A middle ground between the two can be more appropriate in our context.

In conclusion, what we need is a simple flat overlay, with minimal maintenance costs, focused in providing efficient local resource management, but still providing short and locality-aware routing regardless of the physical distance. In the following sections we detail our approach for delivering an overlay with such features.

3.3 Core design

In the previous chapter we reviewed different sorts of topologies for structured overlays, such as hypercubes, trees, butterflies, etc. However, one of the most commonly adopted ones is the ring topology. What makes the ring an attractive choice is its simplicity. Nodes are mapped into a one-dimensional identifier space and basic routing is provided just by linking them in series. Moreover, nodes have a clear range of responsibility over data, and the maintenance of structure and data mappings are minimal and inline with our requirements.

In addition, having a ring structure in place establishes the base line routing which can easily be improved by introducing additional links which serve as shortcuts in the logical space. We refer to them as *longs links*. While strategies for choosing long links can vary, from Chord's strict distributions [155], to hop counts [86] and multi-rings [69], we focus on strategies which do not involve strict criteria or additional structures to maintain. One of the simplest approaches for creating long links is to use Kleinberg small-world networks [85], a common choice in existing overlays [108, 137, 62].

The Kleinberg model is based on a stochastic distribution of links where the probability of having a link at a certain logical distance is inversely proportional to this distance. This means that a node has quite some flexibility in choosing the links as long as they fall globally within this distribution. As we explain below, this flexibility is very important for reducing maintenance and introducing locality-awareness.

In brief, our basic structure relies on the simplicity of the ring overlay along the flexibility of small-world based long links. In order to increase routing options and reduce number of hops, we also opt for bi-directional routing. This choice is relevant for finding alternative routes to reach nodes in case of churn.

3.4 Reducing maintenance

The minimalist approach in design has to be applied also to the maintenance of the overlay. The idea is to remove any background maintenance protocol which generates redundant traffic and, as shown in [62], does not fully guarantee structural invariants. Alternatively, we rely on a lazy approach in which maintenance is managed only at necessary overlay events, such as node joining, application traffic and failure handling.

Similarly to Pastry [145], the messages generated whenever a node joins the network can be used to notify existing nodes for the arrival of the new node. Additionally, during the joining phase a node can receive routing table entries from its neighbors in the ring, given that their distribution of long links is similar. These are not necessarily the final entries, but it is a relatively good start to make the node operative. These entries can be improved further using application traffic.

Most of the time, the overlay will generate lookup messages as a result of application requests. These messages can be used to piggyback information about the overlay members. This information comes from the nodes in the path the message follows. At each hop, a node can add information about itself into the message, but also information about other nodes in its routing table. Similarly, when a node receives such a message, it looks in it for nodes of interest which can improve the distribution of its long links or which can be saved based on other criteria.

This mechanism shares characteristics with gossiping and ant colony inspired overlays. Nodes exchange information about their routing tables in a lazy way, by means of lookup messages which act as ant-like agents crossing the overlay. One of the main advantages of using application traffic is that redundant maintenance traffic is eliminated. Nodes will create connections between them only because it is needed by the application. Hence, if the application does not require communication between distant edges or an edge and the core, these links will not be maintained just for the sake of the overlay.

Finally, maintenance can be done also when handling failures. Our approach is not to try to avoid the effect of failure by proactively checking the health of the links and taking immediate measures. The goal is to be prepared for such situation so that when the failure occurs, the message is either routed using alternative paths, or it is sent to other destinations (replicas). When this occurs, the overlay reactively tries to repair itself focusing mainly on the ring, since the other links are replaced lazily by the mechanism described above.

3.5 Locality-awareness

In the previous chapter we discussed about three principal approaches for introducing locality-awareness in a structured overlay, namely locality-aware neighbor selection, proximity routing and geographic layout. Here we consider how these approaches can be adopted in our targeted overlay without giving up on its simplicity or traffic minimization goals.

3.5.1 Locality-aware neighbor selection

This approach is applied to neighbors (routing table entries) for which multiple options are plausible. In our context, this applies only to the long links and not to the ring neighbors, since the logical criteria on them is very strict. However, in addition to long links we introduce another set of entries which selection is solely based on latency, called *proximity links*. We describe here the selection of both these sets of links.

Long links. The use of logical small-world networks for selecting long links provides us with the flexibility of accepting various options as long as they comply with a predefined distribution. As a node is involved in application traffic paths, it is provided with information about various

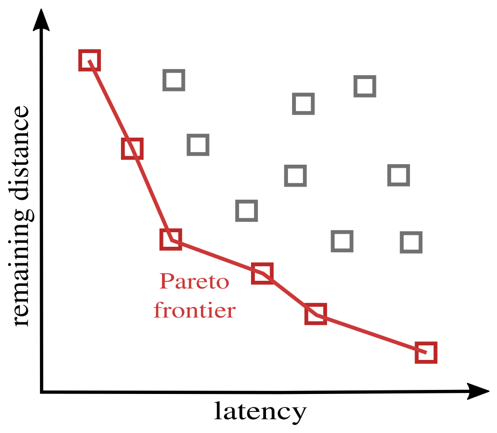


Figure 3.2: Pareto frontier of routing table entries.

other existing nodes. Its goal is to to find low latency long links while still remaining within the bounds of the devised distribution.

In order to evaluate the extent to which a potential long link complies with a distribution, we can adopt the method described in [84], which provides a single value representing the deviation. Then, the idea is to find low latency links while keeping this value below a threshold. However, one problem with this is that we need to provide a latency estimation which can be embedded in the message together with the node identifier.

One way to do this is to adopt network coordinates, which can be used by every node in the system to calculate the latency to a destination without actually starting a communication. However, in our context the maintenance of these coordinates needs to be done in a lazy way, based on the application traffic and their accuracy might be questionable. We discuss about this in the next chapter.

Another way is not to include any latency information in the message. In this case we accept trying new alternative links while still storing the old ones. In this way we lazily test the latency of a certain number of alternatives before settling for the best one within that subset. The main drawback of this approach is that it is based on a local minimum and the sample might still consist of high latency links. However, given the locality of the application traffic, this effect might not be as significant.

Proximity links. The selection of proximity links is very similar to that of long links except the fact that there are no logical criteria. As a node comes across the knowledge of other nodes, it stores them as proximity links if the latency to them is below a certain threshold. This is a straightforward implementation of the local groups described in Discovery (Figure 3.1). The discussion of network coordinates is also valid for the proximity links since they can be used in the selection process. However, another source for proximity links can be also old routing table entries which latency has been tested. These include old ring neighbors or long links, but most importantly links dictated by the application, which we describe below. The local nature of application traffic is favorable in this case too.

3.5.2 Proximity routing

This approach is more lightweight and does not require altering the routing table. The idea is to trade logical greedy routing for less greedy alternatives which include latency considerations. In other words, at each hop a node does not necessarily forward the message to the entry which has the closest identifier to the destination, but it considers other possible entries with lower latencies, provided that they can advance the message logically too. Deciding the entry to select relies on a tradeoff between the logical advancement and the cost of the link.

These alternatives can be plotted in a graph where the Pareto frontier represents the reasonable tradeoffs (Figure 3.2). Determining the best option in the Pareto frontier is an open question which we want to examine, by providing different weights to each of the variables. Our goal is to provide a simple weight setting which can be used to navigate through the Pareto options. Potentially, a policy which assigns these weights dynamically, based on the current state of the overlay or delivery could be devised.

3.5.3 Geographic layout

In the previous chapter we mentioned that the geographic layout could be implemented implicitly or explicitly. The implicit approach requires the identifier of the nodes to be gradually reorganized in order to reflect their proximity. The problem with this approach is that it requires significant coordination between different nodes in the overlay in order to agree about the new identifiers. Additionally, updating identifiers results in additional data transfer, which is undesirable in our context.

On the other hand, the explicit approach requires some prior knowledge about the network which can be embedded in the node identifiers. Nevertheless, if implemented for all the different network levels, this approach results in a static hierarchical structure which we want to avoid for reasons mentioned earlier. However, adopting it just at the edge level without expanding it in higher levels, can still be plausible.

We are aware of the fact that at the edge level nodes are organized in small datacenters (at the PoPs). We can embed this information in the node identifier in order to distinguish between inter and intra-datacenter routing policies, while still keeping a flat topology in the levels above. There can be different ways for nodes identifier to reflect the way they are grouped into datacenters and how the routing policy is affected. We have considered three of them, namely: *single-ring flat*, *hierarchical* and *multi-ring flat* (Figure 3.3).

The single-ring flat approach aims at assigning close identifiers to nodes in the same datacenter, as shown in Figure 3.3a. This is done by splitting the identifier space into different equal-size intervals, each of which is reserved to a single datacenter. The size of these intervals depends on a rough estimation of the maximum number of nodes per datacenter.

This approach has a few advantages regarding maintenance and routing. The maintenance remains simple since no additional structures are introduced. The routing also can be beneficial because a message can progress logically while staying within the datacenter before finding a good hop to an external datacenter. However, there are also some problems with the flexibility of this approach since maintaining reserved splits is difficult and does not scale well, especially when the size is not predefined. Additionally, it requires a larger identifier size for supporting the same system size, which results in a larger diameter ring.

The hierarchical approach consists in splitting the identifier in two parts. One which identifies the datacenter and the other one which identifies the node within that datacenter. In this approach all the nodes in the same datacenter know each other directly. However, inter-datacenter routing is handled by a leader node which acts as a gateway for all the other nodes, as shown in Figure 3.3b (leader in white). The ring neighbors and long links of the leaders are selected based on the datacenter part of the identifier. This approach provides a smaller ring diameter and requires less links to be lazily maintained. Nevertheless, it is not very robust as the leader is a single point of failure and it requires additional mechanisms, such as replication and leader election algorithms, in order to be resilient.

The multi-ring flat approach is very similar to the hierarchical one except that there is no leader. In this case, all the nodes provide inter-datacenter routing (Figure 3.3c). This approach

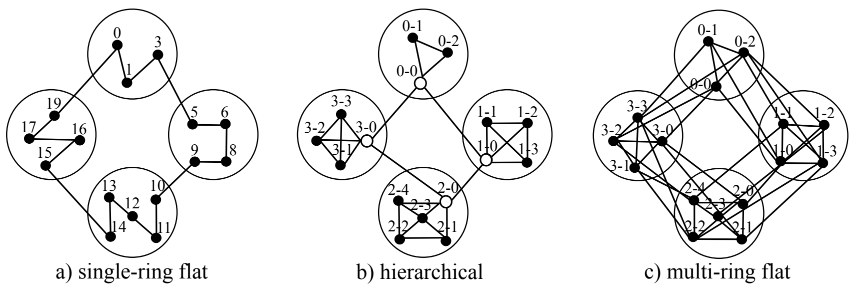


Figure 3.3: Approaches for taking datacenter division into account.

is more robust than the hierarchical one as it distributes the load more evenly. It has the same ring diameter, but it might require more application traffic for each node to find its ideal long links.

This approach enables nodes within the same datacenter to collaborate either by sharing information or assisting in routing. These nodes can share information about the proximity links, since they are similar, or include latencies in internal piggybacking, since in that case the values are relevant. Additionally, similar to the first approach, a node can decide to route locally a message until it finds a good option for sending it to a remote datacenter.

This final approach is more inline with our goals as it keeps the flat structure without introducing any leaders, while its inter-datacenter routing can enables intra-datacenter collaboration. However, the other two approaches can result in more efficient routing in specific conditions. Therefore, we can provide them as flavors of the same overlay.

3.6 Augmented routing tables

Early overlays focused on proving that efficient routing could be delivered with very limited sized routing tables. Initially this was motivated by memory constraints and the presence of heavy maintenance protocols, which discouraged large routing tables. On the other hand, modern zero-hop overlays maintain system-wide routing tables. When the maintenance of such table is done actively, it is not scalable and it generates significant traffic. A reactive maintenance is more reasonable, but it results in nodes storing information which is not relevant in the context of fog, where locality is fundamental. We propose a middle ground between these two extremes, by enriching the routing table we have discussed so far with two extra sets of links: application links and uniform random links.

Applications links enhance our overlay with a semantic layer. They are links dictated by the application's lookup request. They are more dynamic, because they can change over time and over different applications. Given the fact that applications in the fog consist of local traffic, there is a high chance that some of these links have a low latency, therefore they can be converted into proximity links and be stored for a longer period of time.

Random links are used to introduce some additional entropy in our overlay. So far nodes in the same data center have similar ring neighbours, long links, and proximity links, so introducing random links can diversify the routing table of each node. The selection of these links is similar to the one of long links, except that in this case the distribution is uniformly random and the latencies are not taken into account. Uniform random links help creating low diameter random graphs which can improve resilience and security [164] or can enable random dissemination in case we want to support this primitive.

3.7 Conclusion

In this chapter we discussed the design principles of overlays in decentralized cloud management. We focused on devising a simple structure with minimal maintenance traffic, which provides locality-awareness, and minimizes the number of hops. For the core design, we opted for a basic ring structure, with long links following a small-world distribution. For maintenance, we use a lazy method based on piggybacking overlay information within application messages, so that the nodes gradually discover the entries they need to populate their routing table. For the latency-awareness, we made use of all the three approaches. We detailed the latency-aware long links and proximity links selection, and the locality-aware routing based on the trade off between logical distance reduction and low latency links. For the geographical layout approach, we discussed various alternatives for handling nodes in the same datacenter. Finally, we proposed to enhance our routing table with application links, which improve application performance while providing a good source of proximity links, and uniformly random links, which improve the diversity of links in a datacenter.

Chapter 4

Koala: a Lazy and Locality-Aware Overlay for Decentralized Clouds

4.1 Introduction

In the previous chapter, we presented a general high-level view of decentralized management challenges and laid out some design principles for building an overlay network which addresses such challenges. We advocated for a simple flat structure which minimizes maintenance traffic, provides locality-awareness through different mechanisms, and reduces hops by introducing additional entries in the routing table. In this chapter, we focus on providing further details on how these design goals are put together and implemented in a prototype. We introduce Koala, a structured overlay network for efficient service localization which is built upon these principles. We present its architecture details, its algorithms and their validation through a comprehensive set of simulations.

Koala combines existing DHT and passive gossip-like techniques to disseminate information passively without compromising its performance. For this reason, it shares some of the basic concepts with well-known protocols. Nevertheless, Koala revises these concepts and introduces additional ones in order to support lazy maintenance and locality-awareness.

Koala nodes use the Kleinberg's small-world model [85] based on logical distance for filling their routing tables. This model defines the ideal entries of a node's routing table, which deliver a logarithmic hop complexity. However, Koala nodes do not actively search for these ideal entries, but rather fill the routing tables opportunistically, when such nodes come to their knowledge through information piggybacked in the application traffic. In other words, if no traffic is generated by the application, there is no traffic at all within the overlay. Therefore, Koala's performance depends on the amount of traffic generated by applications running on top of it (for instance a cloud stack).

The same lazy approach is applied to failure handling. Koala does not continuously monitor the status of the routing entries in an attempt to proactively repair the overlay in case of failure. Instead, it is designed to handle failures reactively by minimizing their impact through alternative routing, while transparently recovering its structure.

Regarding locality-awareness, Koala focuses primarily in reducing inter-datacenter latencies by making nodes select each routing hop according to a trade-off between a latency-based and a greedy routing based on logical distance. Koala also provides a way to take the multi-datacenter topology into account through integrating the information of which datacenter a node belongs to in its identifier, thus applying different routing policies in intra-datacenter routing.

Our experiments show that despite its lazy maintenance, Koala still delivers a similar complexity to proactive protocols, while it further reduces latencies due to its latency-aware routing. Additionally, they show that passive maintenance is sufficient for repairing the overlay in case of reasonable churn. Finally, we show that Koala adapts well to a multi-datacenter environment.

This chapter is organized as follows. Section 4.2 presents the details of the protocol. Section 4.3 presents our simulation results. Section 4.4 describes how Koala compares to the related work, and Section 4.5 concludes.

4.2 Protocol

In this section, we describe the assumptions on the environment Koala operates. We provide further details on its architecture and present the algorithms for handling overlay creation, routing and maintenance.

4.2.1 Model

We consider a distributed system consisting of nodes which communicate via message passing. These nodes are grouped into different physical locations called Points of Presence (PoPs) and are interconnected so that a message sent from one node can be delivered, in finite time, to any other one. We assume that the latency of a message within the same PoP is insignificant compared to the one between different PoPs. We additionally assume that a communication channel between any two given nodes can be established and that these channels are fair-lossy, meaning that the channels can lose a finite number of messages. Nodes can join or leave (crash) at any time. The crash can be permanent or the node can recover, but then it has to re-join the system. As we aim to use Koala to interconnect nodes in different PoPs, we assume a reasonable level of churn.

4.2.2 Koala: under the hood

In order to explain the basic structure of Koala we do not take into consideration for now the way nodes are grouped in PoPs. For simplicity, we assume that each PoP has only one node. The way the structure is handled when multiple nodes per PoP are considered is a progression of this case and it is discussed in the next section.

Similarly to Chord, nodes in Koala are organized logically in a ring by assigning them unique m-bit identifiers (IDs) from a circular identifier space, as shown in Figure 4.1. However, in contrast to Chord, routing in Koala is bidirectional, meaning that the logical distance d between two IDs is the smallest between the distances calculated in both directions, clockwise and counterclockwise, and it is defined as follows:

$$d(id_1, id_2) = min(|id_1 - id_2|, 2^m - |id_1 - id_2|) \tag{4.1}$$

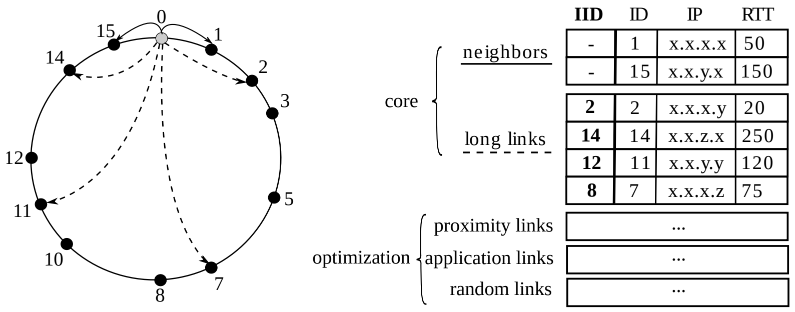
For example, in Figure 4.1 the distance between IDs 14 and 3 is calculated as follows:
$d(14, 3) = min(11, 5) = 5$.

Each node has a routing table consisting of multiple entries. For each entry, a node stores four fields: i) an optional Ideal ID (IID), present only for long links, which represents a target ID for this long link entry (we explain it in more detail below), ii) the logical ID, iii) the IP, iv) the latency to reach that node expressed in Round Trip Time (RTT). Routing entries can be classified in two main groups: *core links* and *optimization links.*

Core links are entries which are essential for enabling efficient routing. They can be either *neighbors* or *long links.*

Neighbors are nodes with predecessor or successor IDs in the identifier space. They ensure the ring structure is maintained so that every node is reachable. For resilience, a node can keep multiple pairs of neighbors, for instance, 2 predecessors and 2 successors.

Long links are entries which enable shortcuts in the ring structure and affect routing significantly. In order to provide a logarithmic routing, long links need to be carefully selected at



IID	ID	IP	RTT
-	1	x.x.x.x	50
-	15	x.x.y.x	150
2	2	x.x.x.y	20
14	14	x.x.z.x	250
12	11	x.x.y.y	120
8	7	x.x.x.z	75
...			
...			
...			

Figure 4.1: Logical ring in a small network (m=4), and the routing table of node 0.

different logical distances from the node itself. In Koala we do this by using a continuous version of Kleinberg distribution which is defined as follows:

$$p(d) = 1/(d \ln(M)) \tag{4.2}$$

This formula defines the probability to have a long link at a distance d, where M is the maximum distance in the ring. Given that Koala is bidirectional, this distance is half of the ring, 2^{m-1}. Note that this distribution favors shorter distances over long ones. As we detail below, we use this probability distribution to generate the ideal distances at which our long links should be. Consecutively, we convert these distances into IDs by randomly either adding or subtracting them from the ID of the node itself. For example, if node with ID=0 in the figure above uses this Kleinberg-based random number generator and gets the number 4, it decides uniformly randomly if node with ID=4 or the node with ID=12 should be in its routing table. The generated IDs constitute the ideal IDs (IIDs) for our entries. Note that generating the IID fields for long links does not mean that nodes with this IID exist. We could actively search for them, but that is against the principle of laziness. Therefore, we rather wait to learn about them (or nodes with close ID to them) in a process we describe later.

Optimization links are additional links which are not strictly necessary for the functioning of the overlay but that can optimize it further. We focus particularly on *proximity links* and *application links*, but other kinds of links, such as *random links* can be used as well in certain situations.

Proximity links are a distinct set of nodes with which a node has communicated in the past and which have a low latency. They are ordered in a fixed size queue based on the RTT value. As entries with smaller latency are added, the ones with higher ones are removed. The combination of long links, which are selected based on their logical distance, and proximity links, which are based on their latency, provides a good input for our latency-based routing algorithm that we will explain shortly.

Application links are entries for which the application has shown interest. Each node maintains a short history of the latest IDs requested by the application and the count of requests addressed to these IDs. When this count reaches a threshold, the source node sends a flagged message which requires the destination to reply back. In this way, the destination is added in the list of application links and further requests are served directly. These links can be useful when such a mechanism is not handled in the application level or when multiple applications have similar patterns.

Random links are entries similar to long links but their distribution is uniform rather than Kleinberg-based. Long links for nodes which are logically close can be similar. Random links IIDs do not depend on the ID of the node itself. Therefore, they introduce more diversity in the possible choices of the routing algorithm.

The number of entries in each group is configurable. For Kleinberg distribution to provide $O(\log N)$ routing (N being the size of the network), we need at least m long links. Therefore,

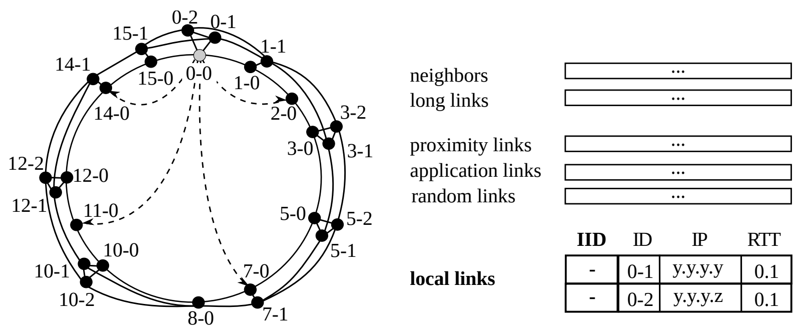


Figure 4.2: Multi-ring structure of the example taken in Figure 4.1.

for long links we choose a multiple of m such as $Nr_{ll} = C_{ll} * m$. We configure the number of long links by setting the value of C_{ll}. Although not necessary for the routing complexity, we can do the same thing with the size of the other groups of links so that it remains relative to the size of the system. Thus, C_n, C_{pl} can control the number of neighbors and proximity links, respectively.

4.2.3 PoP-aware topology

In the previous chapter we anticipated that the PoP structure can be embedded in our overlay in an explicit way, by making the node identifier reflect the PoP it belongs to. We devised three ways to do so, namely *single-ring*, *hierarchical*, and *multi-ring* in Chapter 3. Here we focus on the multi-ring approach since it provides more robustness and does not require extra maintenance steps in case of churn. Note that all the three approaches are available as options in Koala.

In the multi-ring approach, the node identifier is split in two parts, the first part identifies the PoP, and the second identifies the node within the PoP. In the section above, we described the structure of a single ring based on a single identifier. In this section that identifier refers only to the PoP part of the new identifier, while the second part can be chosen arbitrarily, as long as it is unique within the PoP. This means that the distance function used to create long link IIDs applies only to the first part of the identifier. We refer to this a *global distance* or simply logical distance. Placed in a multi-ring setting, the example considered in the previous section is shown in Figure 4.2.

For clarity, we consider only a few nodes per PoP in the figure's example, one to three. While the rest of the links described previously remain the same (except the ID change), a new category of links is introduced: the local links. These are all the other nodes in the local PoP. These nodes could potentially share the same neighbors in the ring, but that is not recommended since it reduces robustness. Therefore, in order to increase the links between neighboring PoPs in the ring, we aim at using distinct neighbors for each node in the PoP. This is done by applying the distance function on the second part of the identifier, which we refer to as *local distance*. Therefore, from all the possible neighbors in the neighboring PoPs, a node selects the one with the smallest distance on the local part of the ID. For example, nodes 1-0 and 1-1 could both be successors of node 0-0, but 1-0 has a smaller local distance, and therefore it is preferred.

4.2.4 Routing: greedy, latency-based, or both?

Once we have defined the full routing table, we need to define a routing algorithm which determines the next hop for a message. The Kleinberg distribution of long links provides us with a logarithmic complexity in terms of number of hops if a greedy algorithm is used. Greedy routing selects the entry which reduces the most the remaining logical distance to the destination. Nevertheless, it does not consider the cost of each hop in terms of latency. As a result, even though the number of hops is reduced, the cost of the total path might still be high. On the other hand, an algorithm which selects the entry with the lowest latency, without considering the logical distance, does not provide correctness and can increase drastically the number of hops. We examine an algorithm which takes into account both factors, logical distance and latency and provides a tradeoff between them. This is done by rating each entry of the routing table as follows:

$$R_{entry} = 1/(\alpha * d(entry.ID, dest.ID) + (1-\alpha) * norm(entry.RTT)) \tag{4.3}$$

where $d(entry.ID, dest.ID)$ is the remaining logical distance if this entry is chosen, $norm()$ is a normalization function which converts the value of RTT into the same scale as the logical distance, and α is a coefficient which determines the weight of each of the factors. An $\alpha = 1$ results in a purely greedy algorithm, and an $\alpha = 0$ in a purely latency-based one. Note that the normalization function maps the minimum and maximum values of the remaining logical distance to those of the RTT, and derives the values in between using a linear function. The remaining distance can vary from 0, when the next step is the final destination, to $d(node.ID, dest.ID) - 1$ if we progress just by one step. For the RTT, the maximum value can be introduced empirically (if the estimation is smaller than the real value, the consequence is the same as decreasing α. Similarly, an overestimation is equivalent to increasing α.

Algorithm 1 Routing.

```
function onRoute(msg)
    if msg.dest $\neq$ this then
        if msg.src = null then
            initializePiggyBack(msg)                     ▷ 4 - 6: explained in Section 4.2.6
        end if
        msg.piggybackRT.add(this.asEntry())
        fwd $\leftarrow$ this.getNextHop(msg.dest)
        if fwd $\in$ this.RT.neighbors.succs then        ▷ 8 - 12: explained in Section 4.2.7
            msg.piggybackRT.add(this.RT.neighbors.preds)
        end if
        if fwd $\in$ this.RT.neighbors.preds then
            msg.piggybackRT.add(this.RT.neighbors.succs)
        end if
        send(fwd, msg)
    else
        notifyApplication(msg)
    end if
end function
function getNextHop(dest)
    maxRating $\leftarrow$ -1, maxRatingEntry $\leftarrow$ null
    for each entry $\in$ this.RT do
        rating $\leftarrow$ getRating(entry, dest)
        if rating > maxRating then
            maxRating $\leftarrow$ rating, maxRatingEntry $\leftarrow$ entry
        end if
    end for
    return maxRatingEntry
end function
function getRating(entry, dest)
    if d(this.ID, dest.ID) $\leq$ d(entry.ID, dest.ID) then
        return -1
    end if
    if d(entry.ID, dest.ID) = 0 then
        return $\infty$
    end if
    return 1 / ( $\alpha$ * d(entry.ID, dest.ID) + (1 - $\alpha$) * norm(entry.RTT))
end function
```

The details of the routing algorithm are provided in Algorithm 1. The piggybacking aspect is detailed later. When a node receives an application message, the *OnRoute* method is called. If this node is the destination, the message is sent to the application, otherwise it is forwarded to the node with the highest rating in the routing table (in *getNextHop*). Note that in the function *getRating*, we discard nodes for which the distance to the destination is not smaller than the one of the node itself (Line 31). These steps provide correctness of the routing even when $\alpha = 0$. Additionally, if the destination is present in the routing table, we forward directly to that entry without taking the latency into account (Line 34).

However, in a multi-ring setting, the condition of always reducing the logical distance regardless of the α, results in messages leaving immediately the PoP. While this is generally a reasonable approach, we would like to explore other cases in which we route the message locally for a limited number of inexpensive hops with the aim of finding a better link which gets closer or faster to the destination.

These can be treated as exceptional cases, such as when the selected entry has still a rating below a certain threshold. When this occurs, a policy can be to send the message to a random local node, together with the best local entry piggybacked. This can be repeated a few times until a link with a good rating is found, or the best of the explored nodes is chosen. However, other policies can be devised too. Our interest is to see if a local hop can be beneficial to the global routing.

4.2.5 Joining

The joining process is slightly different for a node that initializes a PoP (the first node of this PoP to join the overlay), and for nodes who join in an already existing PoP. We first consider the joining of the first node in a PoP.

As detailed in Algorithm 2, before joining the network a node generates its own unique ID, consisting of the two parts: the new PoP ID and a default "0" local ID. The PoP m-bit ID is generated by hashing its IP (Line 3). Based on this ID it initializes its long links. As we explained before, for each link, we generate random distances according to Kleinberg distribution and convert them into IIDs by randomly adding or subtracting them from the ID of the node. Note that initially, the ID, IP and RTT fields are set to some default values (Line 18). ID_{df} is a special value such that its distance from any other ID is infinite: $d(ID_{df}, *) = \infty$. IP_{df} is an empty string, whereas RTT_{df} is set to an approximate value of the average RTT in the system. Note that random links are initialized similarly, except that uniformly random distances are generated in Line 17. After the initialization phase, the node tries to join the network by sending a request to an existing node (commonly called the *bootstrap* node) which IP address is known beforehand (Line 12).

This request triggers the *onExchangeRT* handler on the recipient node. The latter examines the received routing table, which includes all parameters passed to the *ExhangeRTMsg* constructor, and checks if there are neighbors or links of interest. That is, a link which ID is closer to the IID of any of the current links than the ID of the link itself. If such a link is found we update our current links as shown in method *updateLinks*. Since the sender just joined, the received table consists of only the sender. Assuming that the first recipient is not an immediate neighbor of the joining node, the request is forwarded to an entry in the routing table (Line 43). Forwarding will continue until a neighbor of the joining node is reached.

The neighbor will reply to the joining node by sending its routing table, its own entry details and its old neighbors (Line 38). The joining node, would look through the received entries and update its long links (generally copy them) and also find the other immediate neighbor, which was the old neighbor of the sender. Consequently, the joining node would contact it as well and receive from it its routing table which will be used to update its long links. Note that an extra message is sent to the first discovered neighbor but that can be avoided by specifying a flag for the exchange request which distinguishes joining requests from the others. As a result of these exchanges, the joining node will find its correct position in the ring and will have a list of links by merging the routing tables of its immediate neighbors.

When a node joins the network in an already existing PoP, it uses one of the other nodes in that PoP as bootstrap. In this case, the joining node asks the bootstrap for its ID (Line 6).

Algorithm 2 Joining.

```
 1: function onJoin(newPoP)
 2:   if newPoP then
 3:     id = Hash(this.IP, m) + "-0"
 4:     this.onReceiveID(id)
 5:   else
 6:     GetID(this.bootNode)          ▷ asks bootnode for ID and gets back control at onReceiveID
 7:   end if
 8: end function
 9: function onReceiveID(id)
10:   this.ID = id
11:   this.intializeLongLinks()
12:   send(this.bootNode, ExhangeRTMsg(this.RT + [this.asEntry()]))
13: end function
14: function intializeLongLinks( )
15:   this.RT.longlinks ← [ ]
16:   for i = 1 → $C_{ll} * m$ do
17:     newIID ← this.getKleinbergIID()
18:     newLL ← Entry($ID_{df}$,$IP_{df}$, newIID, $RTT_{df}$)
19:     this.RT.longlinks.add(newLL)
20:   end for
21: end function
22: function getKleinbergIID( )
23:   d ← round($2^{RandomDouble(0,1)*(m-1)}$)
24:   if RandomInt() % 2 = 0 then
25:     return (this.ID+d) %$2^m$
26:   end if
27:   return (this.ID-d+$2^m$) %$2^m$
28: end function
29: function onExchangeRT(msg)
30:   oldNeighbors ← this.getImmediateNeighbors()
31:   for each entry ∈ msg.srcRT do          ▷ srcRT contains parameters of ExhangeRTMsg
32:     updateLinks(entry, this.RT.longlinks)          ▷ omitting proximity links
33:     updateNeighbors(entry)
34:   end for
35:   newNeighbors ← this.getImmediateNeighbors()
36:   for each entry ∈ newNeighbors do
37:     if entry ∉ oldNeighbors then
38:       send(entry, ExhangeRTMsg(this.RT + [this.asEntry()] + oldNeighbors))
39:     end if
40:   end for
41:   if msg.src ∉ newNeighbors then
42:     fwd ← this.getNextHop(msg.src)
43:     send(fwd, msg)
44:   end if
45: end function
46: function getImmediateNeighbors( )
47:   return [this.RT.neighbors.succs[0], this.RT.neighbors.preds[0]]
48: end function
49: function updateLinks(newEntry, rt)
50:   for each entry ∈ rt do
51:     if d(entry.ID, entry.IID) > d(newEntry.ID, entry.IID) then
52:       entry.ID ← newEntry.ID
53:       entry.IP ← newEntry.IP
54:       entry.RTT ← newEntry.RTT
55:     end if
56:   end for
57: end function
```

The bootstrap generates a new ID based on its PoP ID and a local ID which is not present in its local links, and after broadcasting the information of the new node locally, it returns the ID to the joining node which gets the control in *onReceiveID* method (Line 9). At this point the joining process continues as before, but in this case the ring neighbors are received from the local node. These neighbors might be eventually updated after contacting them and learning about other neighbors in the same PoPs with smaller local distances.

4.2.6 Lazy learning

As described above, a node learns from its neighbor about links of interest during joining. However, learning does not stop there, but it continues permanently as messages are routed. Each routed message is provided with a data structure very similar to the routing table of the nodes (the *msg.piggybackRT*). At each step during routing, this structure is enriched with information about existing nodes. One source of this information is the path of the message. Whenever a node forwards a message, it piggybacks on it its own details as shown in Line 6 of Algorithm 1.

A second source of piggybacked information is the routing tables of nodes in the message's path. The source node which starts the routing request initializes also the piggyback structure of the message as shown in Algorithm 3. This is very similar to the initialization of the long links except that the distribution is rather uniform. This is done to improve the diversity of the information coming from the first source as it does not depend strictly on the path itself. We keep the size of the piggyback structure logarithmic to the size of the network as we do with all the other structures using the C_{pb} setting. The value of C_{pb} is chosen based on a tradeoff between fast learning and message overhead.

Algorithm 3 Learning while routing.

```
function intializePiggyBack(msg)
    msg.piggybackRT ← [ ]
    for i = 1 → $C_{pb} * m$ do
        newIID ← RandomInt($2^m$)
        newP ← Entry($ID_{df}$,$IP_{df}$, newIID, $RTT_{df}$)
        msg.piggybackRT.add(newP)
    end for
end function
function onReceiveMsg(msg)
    for each piggyEntry ∈ msg.piggybackRT do
        updateLinks(piggyEntry, this.RT.longlinks)        ▷ omitting proximity links
        updateNeighbors(piggyEntry)
    end for
    for each entry ∈ this.RT do
        updateLinks(entry, msg.piggybackRT)
    end for
end function
```

Whenever a message is received, the *onReceiveMsg* handler is called. Initially the receiver node will look in the piggybacked structure for links of interest and neighbors as done during routing table exchanges. At a second step, it will try to update this structure with links from its own routing table. Therefore, at each step, a node receives information about the nodes present in the network and distributes also its own knowledge. These simple mechanisms can be thought of as passive gossiping. Each node gossips information about the state of the network, but without periodically contacting any node solely for this purpose. As a result, the speed of learning about the network depends directly on the amount of application traffic.

4.2.7 Resilience

The extent to which a protocol is resilient to changes in the network depends on its ability to maintain the links in routing tables. However, periodic maintenance procedures are not compatible with our main principle, which is being as lazy as possible. For this reason, our protocol is not designed to mask the effects of churn as soon as it occurs. It rather copes with churn by reactively restoring the overlay, without degrading the performance.

Node joining and departing are handled differently depending on whether they are neighbors or long links. In case they are neighbors, taking action immediately is much more important than when they are long links. As described before, when a node joins the network, its future neighbors are immediately notified. On the contrary, upon a node departure, its neighbors are not notified until one of them tries to contact it. At this point, there is a difference between when a single node departs and when the whole PoP is not reachable.

When a node realizes its neighbor is not responding, it asks a certain number of local nodes to check if they can reach their neighbors in the same PoP. If it gets a positive response, it updates its link to the new working neighbor. However, when none of the checks succeeds, then the node assumes the whole neighboring PoP is down.

At this point, the node will exchange routing tables with the next node in its neighbors list so they become immediate neighbors. For this to work, the list of neighbors needs to be up to date to some extent. We do this by piggybacking neighbor information when messages are exchanged between neighbors. For instance, when node q sends a message to its successor s, it also reports its predecessor p. So in case q departs, s and p will exchange routing tables. This is shown in Lines 8 - 12 of Algorithm 1.

This lazy mechanism for knowing neighbors' neighbors does not guarantee that the information will not be outdated. In case a node loses all its neighbors in a certain direction (for example, all the predecessors), it will try to search for its new neighbors by contacting a working long link in the vicinity (possibly in the direction of the lost neighbors). This is a special search because it targets nodes between the long link and the node, therefore it always follows the same direction in the ring. If no intermediate nodes are found, then the long link becomes the new neighbor.

When a node restores the ring by connecting to a neighbor in a new PoP it also receives its local links from it. The node then broadcasts this list locally so that all the nodes in the same datacenter can find their own neighbors based on the shortest local distance.

In case a long link joins or departs, the management is more straightforward. Joining nodes which might be ideal long links for current nodes are noticed only as information gets disseminated using piggybacking. A node will realize that a long link is down only when trying to forward a message to it. In that case, it will forward the message to the link with the second highest rating according to Equation 4.3, and it will mark this link as down. That means that this link can be replaced with other links, even with a higher difference form the IID. In order to avoid re-adding links, we timestamp each link using Lamport clocks [92].

When the rating of all entries in the routing table are smaller than the one of the node itself, the node declares failure to the sender of the message. This happens only in cases of extreme churn, when all the nodes in the routing table with a closer ID to the destination are down, including neighbors, which can be temporarily unavailable during the neighbor searching phase.

4.2.8 Discussion: Network coordinates

As mentioned at the beginning of this section, we use the RTT of previous interactions to have an estimate of the latency to a node. This estimation is valid as long as it stays within the node's PoP. However, when distributing this information through piggybacking to nodes in different PoPs, it becomes irrelevant. Without using latency information in messages, it would become difficult for nodes to select entries with low latency from piggybacked information. This means that nodes should first accept the information by assigning discovered nodes a default latency value, test it, and then discard it if it is not good enough. The problem with this is that nodes might probe many useless links which turn out to have a high latency, and might miss low latency options which they never tried.

Using network coordinates eliminates to some extent this problem, because the coordinates do not have a single point of reference. They are useful to any node in the system, and can be easily piggybacked with the node information. In this case, selecting nodes from parsed messages is more straightforward because the latency can be estimated by computing the difference of coordinates, without the need for probing them. This results in a faster and less expensive collection of low latency links.

The issue with network coordinates is that in order to have good estimation, sufficient random traffic between nodes is required. This might be not optimal when using it in Koala, as our overlay relies only on application traffic, which follows a certain pattern that might yield imprecise coordinates. If adopting network coordinates requires constantly exchanging messages, this goes against our traffic minimization policy and might be as expensive as the probing discussed above.

In Koala we have integrated network coordinates based on Vivaldi [41], by computing them lazily when interactions between nodes occur due to the application requests. However, our experiments rely mostly on random traffic. To generalize this result, real applications' traffic patterns should be explored.

4.3 Evaluation

In order to evaluate our overlay, we have conducted various experiments using the PeerSim [122] simulator. For each experiment, we use the same basic setup. We assign random coordinates to each PoP on a unit 2D-coordinate system. Nodes within the same PoP have the same coordinates. We use the Waxman model [166] for creating links between these PoPs. In this model the coefficients α and β are selected in such a way that the model respects roughly the neighbor degree of Renater[1], a research ISP network in France. This model does not guarantee a connected graph at once. Therefore we recursively apply Waxman on randomly picked nodes from the connected sub-graphs until the graph is connected. The resulting topology represents our *physical network*. In order to simulate the IP routing in the physical network, we use Dijkstra shortest path algorithm [44], where the cost of an edge is a function of the Euclidian distance between the nodes. The cost of a physical path represents its RTT. Consequently, the RTT of a logical path in Koala is the sum of RTTs of all physical paths composing it.

In order to compare Koala with physical routing or other protocols, the simulator assigns the exact same list of tasks to all protocols. A task can be: routing a message, adding a node, or removing a node. At the end of a task, metrics such as RTT, physical and logical hops, and failures are collected and compared. Although we use PeerSim in the event-based mode, tasks are processed on a cyclic basis. At each cycle one or more tasks can be processed. Our reported results are based on the average values of a group of cycles. We show the size of this group in round brackets in the X-axis label.

We conduct five sets of experiments, presented in sections 4.3.1 to 4.3.5, which focus on: Koala's scalability, latency-awareness, reliability, topology-awareness, and adaptation to different traffic patterns. The first three sets concentrate on Koala's inter-PoP aspects, and therefore we assume each node belongs to a different PoP, *i.e.* we have one node per PoP. However, in the last two sets, we focus on topology-awareness, and therefore we consider multiple nodes per PoP.

4.3.1 Number of long links and scalability

In the first experiment, we study the behavior of our overlay as the network scales up. Following a uniform traffic distribution, at each cycle, one node joins the network and also a message is sent from a random source to a random destination. We scale up to 100K nodes and assume no node leaves the network. We run the same experiment for different numbers of long links, by varying the constant C_{ll}. We fix α to 1, therefore routing does not take the RTT into account. As for the rest of the experiments, the number of neighbor pairs is set to 2 ($C_n = 2/m$, 2 successors, 2 predecessors), and the piggyback list size to m ($C_{pb} = 1$). In this case proximity, application and random links are not considered ($C_{pl} = C_{al} = C_{rl} = 0$). Figure 4.3 shows how latency and logical hops are affected as the network scales up (physical hops are omitted as we focus on the comparison of logical ones).

[1]https://www.renater.fr

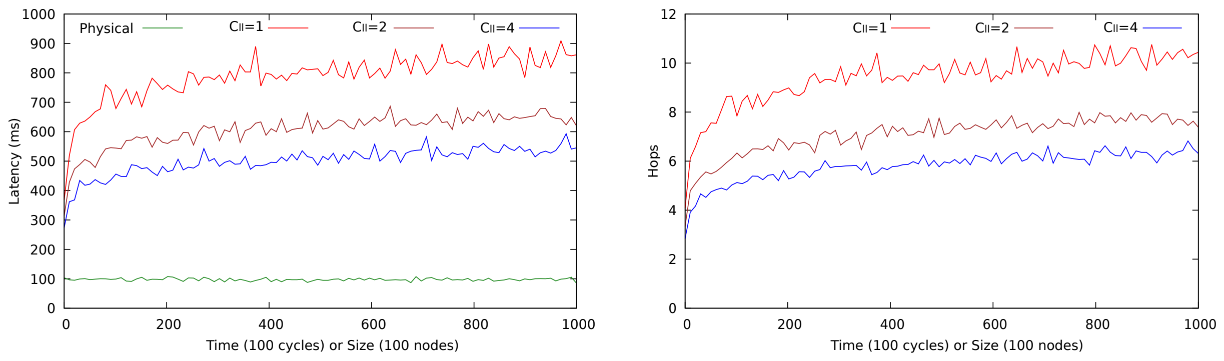


Figure 4.3: Latency and number of hops as network scales up to 100K nodes.

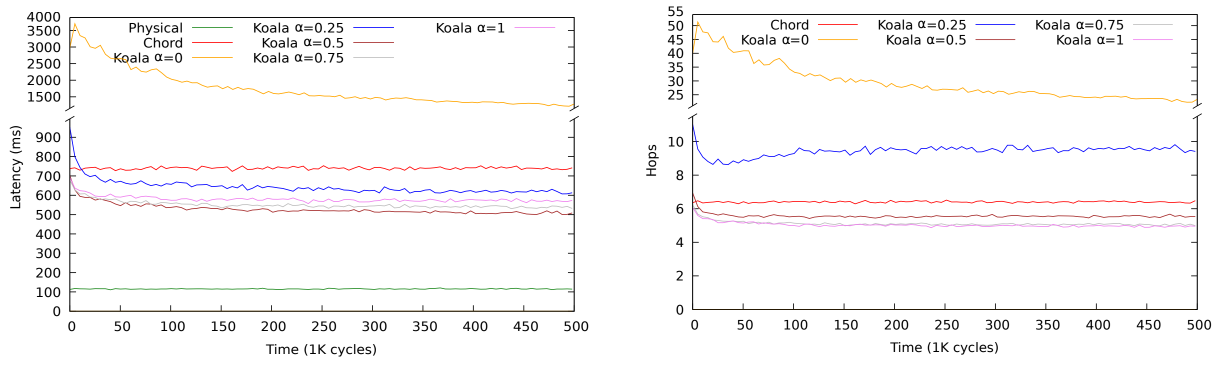


Figure 4.4: Impact of α on latency and hops.

As mentioned in section 4.2.2, $Nr_{ll} = C_{ll} * m$. For $N = 100K$, $m = 17$, therefore $C_{ll} = 1, 2, 4$ correspond to $17, 34$ and 68 long links. Note that one point in the graph represents the average of 100 routes. In Figure 4.3, we observe that regardless of the number of links, both latency and hops grow logarithmically as more nodes join. It is clear that the higher the number of long links, the lower the latency and the hops. However, by doubling the number of links, we do not reduce twice the latency. Therefore, at some point, increasing further the number of long links would not provide significant benefit, while it might introduce additional latency in case of multiple attempts to connect to stale long links.

4.3.2 Lazy learning and latency-awareness

In the following experiment, we demonstrate the ability of Koala to discover the network in a lazy way, as application messages are routed. We particularly focus on the impact of the parameter α on latency and hops. We use a static network of 10K nodes, where no nodes join or leave the network. For this experiment we compare our protocol with Chord. At each cycle, both protocols are requested to route a message from the same random source to the same random destination. As in the previous experiment, we fix the number of long links by setting $C_{ll} = 2$ (28 long links) and we do not consider the other kinds of links links. For Chord we use 4 successors (equal to Koala's neighbors). We repeat this experiment by varying α from 0 to 1, by adding each time 0.25. Figure 4.4 shows the impact of this variation on latency and number of hops.

We observe that regardless of the value of α, as more messages are routed, Koala nodes discover long links which are closer to their ideal ones, thus latency and hops decrease over time. Note that this learning is faster at the beginning. As the discovered long links comply more and more with the Kleinberg model, finding the exact ideal link does not result in significant latency improvements.

Concerning the impact of α on latency and hops, we notice that for $\alpha = 0$, when the routing decisions are almost not at all based on logical distance, Koala delivers a poor performance. This is because at each hop a message is always forwarded to nearby nodes without significantly approaching the destination, which leads to many hops and as a consequence, in very high latencies. However, if we consider slightly more the logical distance ($\alpha = 0.25$), we still have a relatively high number of hops, but we significantly improve the latency compared to when $\alpha = 0$. On the other hand, for $\alpha = 1$, when routing is based solely on logical distance, we achieve the lowest number of hops, but that does not mean the lowest latency. To improve latency, one needs to take RTT more into account by reducing α. This can be done until an equilibrium is found where reducing it more results in higher latency as the number of hops also increases. For our simulation, this equilibrium happened to be for $\alpha = 0.5$, but this value depends on various factors such as network topology and traffic pattern. In addition, we notice that for this setting Koala delivers messages up to 32% faster than Chord. This is not only because Koala uses more long links (accountable for 65% of the gain when $\alpha = 1$), but also due to its latency-awareness (accountable for the additional 35% of the gain when $\alpha = 0.5$).

So far we have considered only links selected on their distance criteria, and therefore the tradeoff routing function is mainly determined by the logical distance. We continue the previous experiment by considering proximity links too. We do not change the overall number of links, instead we divide them equally between long links and proximity links. Therefore, we compare using $C_{ll} = 2$ with $C_{ll} = 1$ and $C_{pl} = 1$. We do this for the best setting from the previous experiment, when $\alpha = 0.5$ and fix this setting for the rest of the experiments.

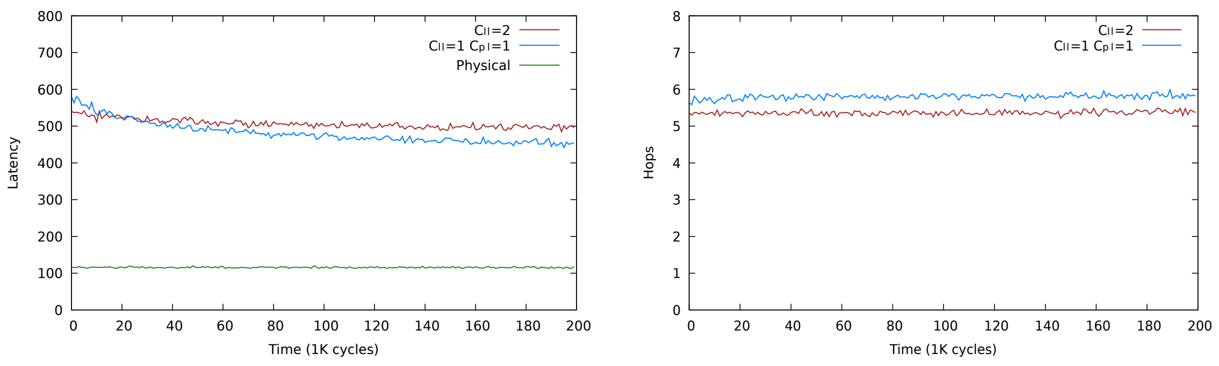


Figure 4.5: Impact of proximity links on latency and hops.

In Figure 4.5 we observe that when combining long links and proximity links, the number of hops is slightly higher than when using only long links. However, as more and more proximity links are found the routes are more latency-aware and the delivery time is improved (by additional 10%).

4.3.3 Resilience to churn

In this set of two experiments we study Koala's ability to lazily repair itself in presence of failures. In the first experiment we consider a network of 5K nodes. As before, at each cycle a message is routed between a random source and a random destination. However, every 80 cycles, CHURN nodes leave the network, and the same number join. These two events do not necessarily happen in the same cycle. In all the experiments in this section we use $C_{ll} = 2$ and no other links as we mainly focus on keeping the Kleinberg properties under churn. We analyze how Koala deals with various levels of churn by varying the number CHURN.

Figure 4.6 shows the effect of churn on latency (right side) and failure rate, *i.e.*, percentage of messages that failed to reach their destination (left side). In the left figure we notice that, as the level of churn increases, the routing latency of successfully delivered messages increases as well,

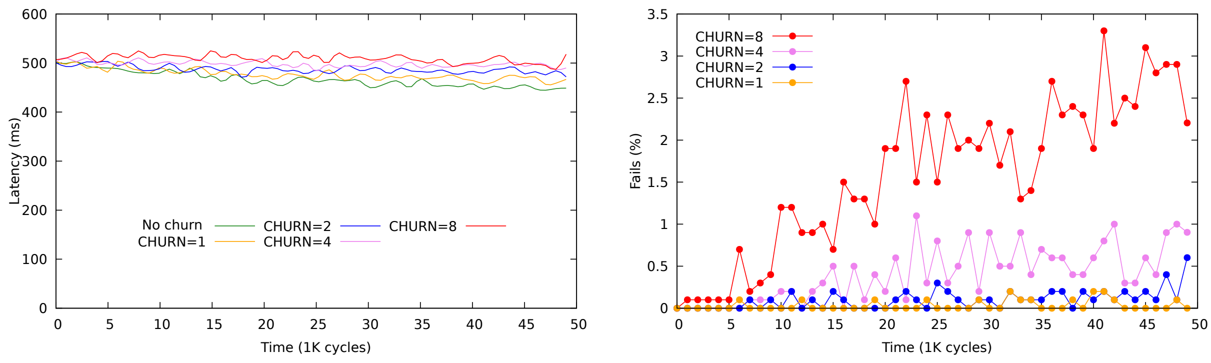


Figure 4.6: Impact of churn on latency and failure rate.

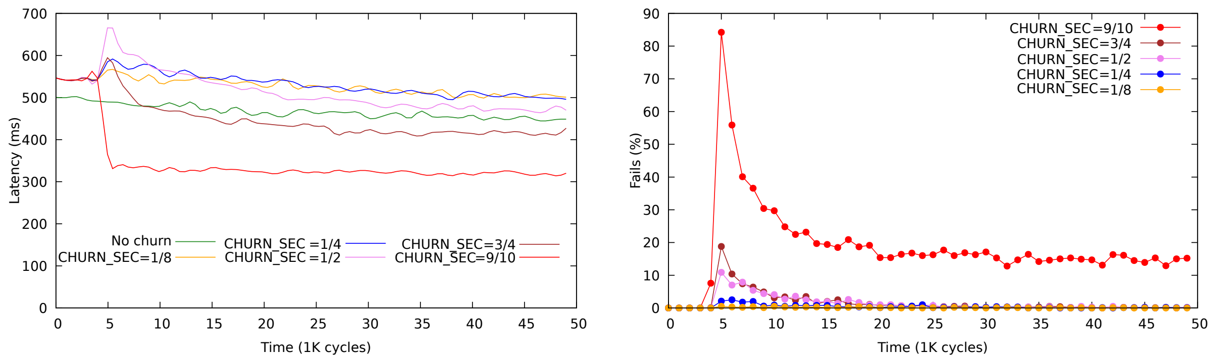


Figure 4.7: Impact of a massive failure on latency and failure rate.

but rather gracefully. This is due to our learning techniques which help nodes to update their stale links without significantly breaking the Kleinberg distribution. Nevertheless, the same thing cannot be said for the number of failures. The right figure shows that for low levels of churn (CHURN = 1 or 2), the lazy repairing is efficient enough to keep failure rate low. However, as churn levels increase (CHURN = 8), the overlay does not process enough traffic per unit of time in order to repair before other changes in the network occur. In that case the failure rate continuously grows, but note that this is an extreme level of churn.

In the second experiment we analyze the ability of Koala to repair itself in case of an abrupt failure of a whole section of the network (CHURN_SEC). In this case, we consider a network of 10K nodes with no joins. At each cycle, we route as in the previous experiments, but at cycle 5K we introduce the unexpected failure. We vary the size of the failed section and show its effects on latency and failure rate in Figure 4.7.

As the amount of long links per node affected by this sort of failure is significantly higher than in the previous experiment, the Kleinberg distribution is initially broken, and this results in higher latencies. However, as the ring starts repairing itself and the number of nodes is reduced, the latencies are reduced as well. Nevertheless, the right side of Figure 4.7 shows that the ability of Koala to repair depends on the amount of failed nodes. We can observe that the higher this amount, the longer it takes for the overlay to rebuild itself. Moreover, if this section of the network is too large (90%), Koala is not able to fully recover anymore, but this is rather unrealistic.

4.3.4 Topology-awareness: Koala flavors

So far, we have considered a topology where each PoP consists of only one node. In the following two experiments we examine the performance of Koala when multiple nodes per PoP are present. For this scenario, we have provided three different approaches (flavors) for taking the

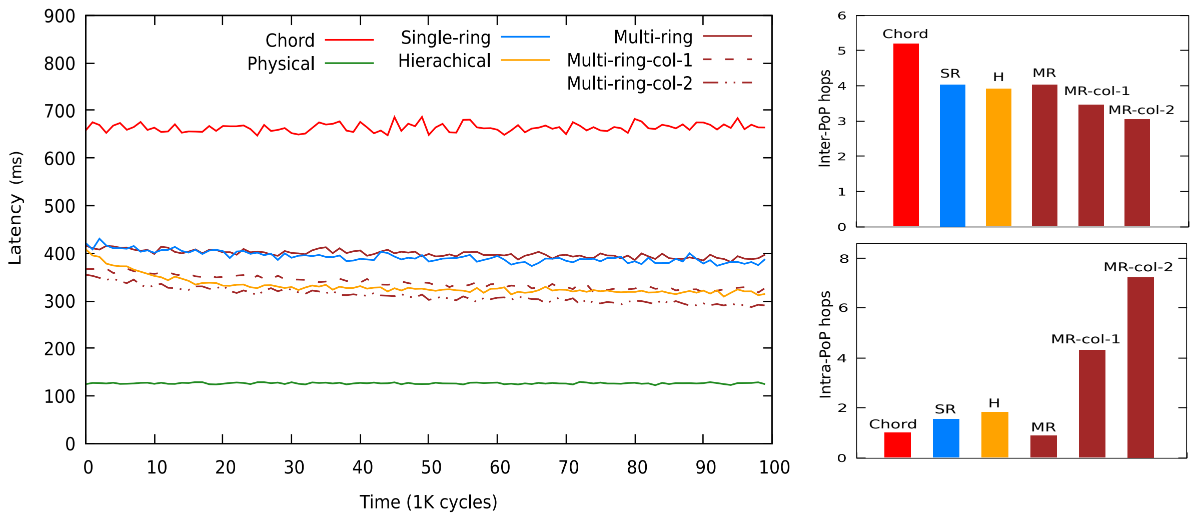


Figure 4.8: Comparison of Koala flavors.

PoP structure into account: single-ring (SR), hierarchical (H), and multi-ring (MR). While the structural and qualitative differences between these approaches were explained in Section 3.5.3 of the previous chapter, in this experiment we focus on their performance comparison.

As mentioned earlier, the multi-ring version of Koala enables local collaboration between nodes in the same PoP so that they assist each other in global routing. Therefore, in this experiment we also consider a collaboration policy. In this policy, whenever a node needs to send a message to an external node (in another PoP), it first routes the message locally, by piggybacking its best destination. The message does a fixed number of local hops, after which the best option (from the cumulative routing tables) is selected for the inter-PoP hop. In our experiment, we consider three cases of multi-ring: when no collaborators are used, when the message is sent locally only once (1 collaborator), and when it is sent twice (2 collaborators). In addition, to the three flavors and the collaboration options, we also include a version of Chord in which nodes in the same PoP are given consecutive identifiers, similar to the single-ring version of Koala.

For this experiment we consider a network of 10K nodes divided in 1K PoPs (10 nodes per PoP). As in Experiment 4.3.2, the network is static, and at each cycle a message is sent from a random source to a random destination. We set the number of long links and proximity ones with $C_{ll} = 2$ and $C_{pl} = 1$, while the rest of the settings are as in the previous experiments. The average latency, the number of inter-PoP hops, and intra-PoP hops are shown in Figure 4.8.

We notice that even though the last few hops of Chord might be local (intra-PoP), the number of inter-PoP hops is still quite higher than with any of the versions of Koala and that is also reflected in its latency. Between the Koala approaches, we notice that the hierarchical one achieves a better performance than the single-ring and the multi-ring without collaboration. This is due to the fact that leaders process a high amount of traffic which allows them to learn faster about their ideal links and provide a better latency. The other two approaches seem to provide a very similar performance, although the single-ring provides slightly more local hops. These local hops in this case are not used to collaborate, but simply to progress logically in the ring, and thus they do not provide a significant benefit. When the collaboration is enabled in the multi-ring, we notice that the increase of the local intra-PoP hops results in a decrease of inter-PoP ones, and therefore to the latency. In the case of two collaborators, the multi-ring approach outperforms the hierarchical one, while being completely decentralized. Thus, in the next experiment, we fix the number of collaborators to two for the multi-ring approach.

4.3.5 Traffic patterns: application links

In all the previous experiments, we have considered only uniformly random traffic, *i.e.* the source and destination of a message were chosen uniformly at random. In reality, traffic tends to follow more specific patterns. Even though we do not target a specific application pattern, we distinguish between three kinds of messages: local, close and global. Local messages have their source and destination in the same PoP. Close messages are those whose latency to the destination is below a certain threshold. In our case, this threshold is taken to be 25% of the largest latency in the network. Messages whose latencies to the destination are above this threshold are considered to be global.

In this experiment, we consider a few traffic distribution options, namely: mostly local, mostly close and mostly global, as shown in Table 4.1. In each case, we focus on a particular message type by giving it a high occurrence (80%), in order to highlight the impact it has on the protocol. As in the previous experiment, we examine the three flavors of Koala, but for the multi-ring we consider only the case when two collaborators are used.

Given that in this case traffic follows certain patterns, application links might impact the performance of our overlay. Therefore, we analyze this impact by using two settings for the routing table links. The first setting is the one of the previous experiment: $C_{ll} = 2$, $C_{pl} = 1$, where application links are not used. The second setting is: $C_{ll} = 1$, $C_{pl} = 1$, $C_{al} = 1$, where we convert half of long links into application links, while keeping the overall size of the routing table unchanged. Figure 4.9 shows the progression of latencies for all Koala flavors and Chord in the three different traffic distributions (shown in rows) in each of the two settings (shown in columns).

In the first column, where the setting is as in the previous experiment, we notice that the three flavors have relatively similar performances to the ones in the previous experiment, where the traffic was uniformly random. Except for the case where the traffic is mostly local (1.a) in which their performance is almost the same, in the other two cases the hierarchical and multi-ring approach are relatively better due to their ability to choose good non-local hops (either through good long links or collaboration). However, in the second column, when application links are used, we notice that the three flavors react differently to the various traffic distributions. Especially for mostly close traffic (2.b), we notice that although all of them benefit from the application links, the multi-ring approach performs significantly better than the other two. The reason for this is that in the multi-ring approach the application links discovered by each node in a PoP are accessible to all the other nodes of that PoP through collaboration. When the traffic is mostly global, the application links are slightly less effective. This is due two a couple of reasons. First, since application links are a source of proximity links, when the traffic is global, less proximity links are discovered. Second, given that the number of close nodes is smaller than that of the global ones, the probability of reusing global application links is lower than that of reusing close ones. In any case, we can say that the multi-ring emphasizes the positive impact certain links have on the performance, due to the local information sharing.

	Local	Close	Global
a) Mostly local	80%	10%	10%
b) Mostly close	10%	80%	10%
c) Mostly global	10%	10%	80%

Table 4.1: Traffic distributions

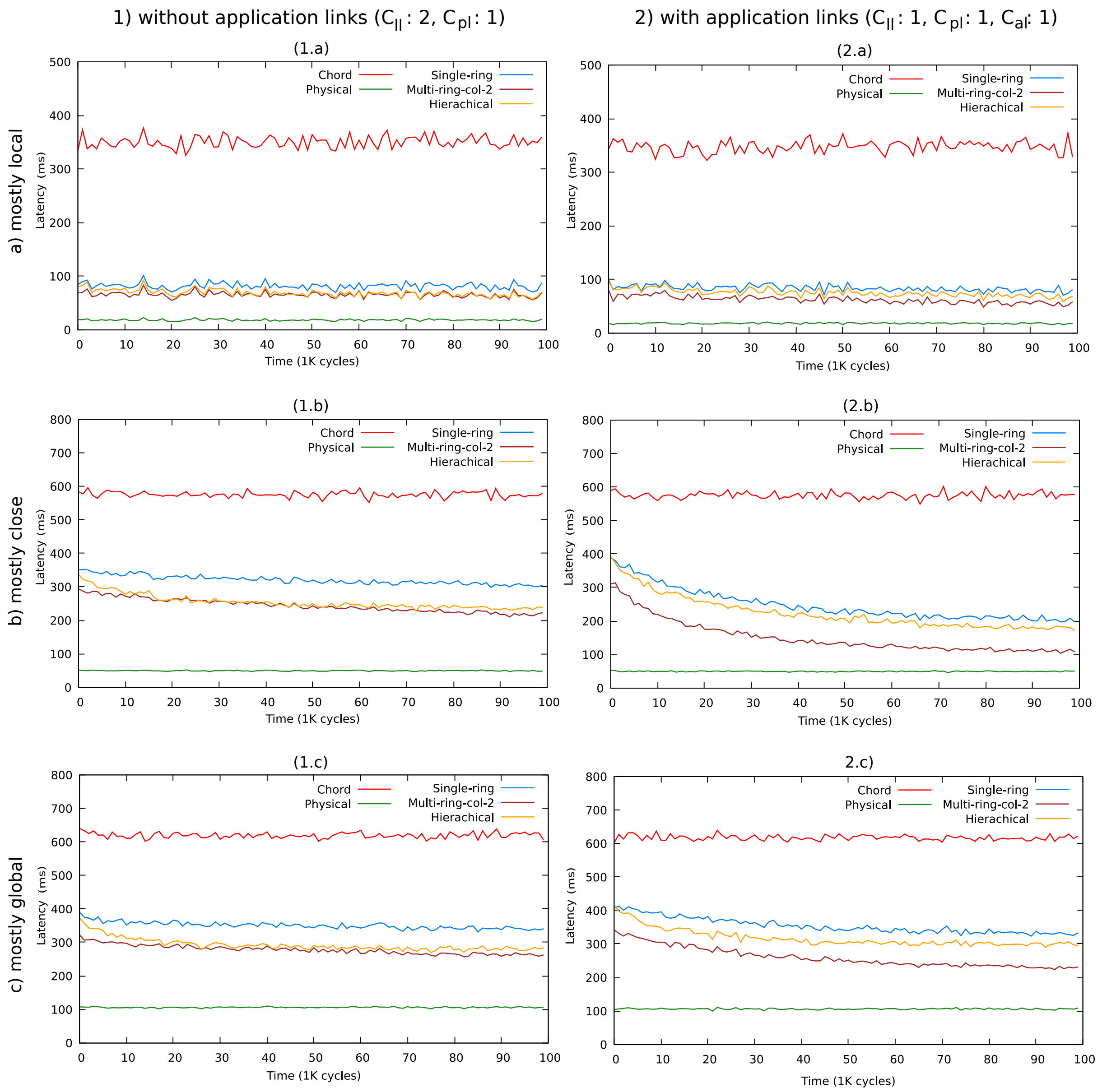


Figure 4.9: Traffic distribution and impact of application links.

4.4 Comparison to related work

As a structured overlay, Koala shares many core aspects with similar overlays, such as Chord and Symphony. However, it does not have any background mechanism for detecting and repairing the routing table. Problems are detected and repaired in a lazy way, only when routing application traffic by means of piggybacking. Piggybacking is used also in Kademlia, but only partially, to enable multiple routes between the same source and destination. The concept of using application traffic for building overlays is used also in [87]. This study assumes that the application issues repeatedly similar queries, and therefore it links together nodes which collaborated in previous queries. This means that the overlay is adapted to a specific application. Koala instead, uses application traffic to optimize the overlay independently of the applications running on top of it. Relax-DHT [94] follows a lazy approach on data replication, by allowing replicas to get further from the root as the network evolves. In this way it avoids the need for a systematic rearrangement of data blocks whenever a node joins or leaves. However, this technique reduces data maintenance rather than overlay maintenance as in Koala.

Fuzzynet [62] is one of the overlays which most resembles Koala as they are based on similar principles. It aims at removing any maintenance protocols by reducing this task only to node joining events, and uses small-world networks for selecting the long links. However, differently from Koala, Fuzzynet discovers long links by using heuristic-based peer sampling, which is not lazy. Additionally, it does not maintain a ring structure, but instead relies on data replication in a specific region of the identifier space. This not only makes the joining phase relatively expensive in terms of data transfer, but also provides only probabilistic resource localization.

In this chapter we did not quantitatively compare the laziness of Koala with respect to other proactive overlays as this depends on certain parameters which can vary, such as the periodicity for gossiping or for running the maintenance protocol. We rather focused on the conceptual differences and show that despite being fully lazy, Koala can perform at least as good or even better than non-lazy protocols.

4.5 Conclusion

Koala is an overlay network designed to target decentralized cloud environments. It provides efficient, locality-aware service localization while eliminating unnecessary maintenance traffic. Koala adapts the overlay maintenance to the activity of the applications running on top of it by piggybacking network information in application messages. Therefore, the more active the application is, the more maintained the overlay is. When the application stops communicating, so does the overlay.

Our experiments show that, despite its laziness, Koala still guarantees a similar performance to other traditional protocols by providing a $O(\log(N))$ complexity. They suggest that by finely tuning the degree of locality-awareness and introducing proximity links, we can significantly reduce inter-PoP latencies. Additionally, experiments on failure handling demonstrated that Koala can lazily repair the overlay even under relatively high levels of churn. Finally, we show that in case of PoP-aware Koala flavors, collaboration has a significantly positive impact on latency since more network knowledge is made available to local nodes in a PoP at a low network delay and maintenance cost.

Chapter 5

Core/edge deployment of legacy applications

5.1 Introduction

In chapters 3 and 4 we focused on designing fog overlays in a bottom-up fashion, starting from the infrastructure requirements and the general principles of the fog computing paradigm. In this chapter, we provide a rather top-down client perspective. We consider the practical aspects of fog deployment of an application and aim at identifying further requirements that can transform our overlay into a useful tool that facilitates fog deployment and enables the application to take further advantage from the fog infrastructure.

Edge applications take advantage of the low latencies between the users and the edge resources [97]. However, these resources have limited capacities and do not have the reliability guarantees of the ones in the core. Therefore, we often deal with a *hybrid* core/edge deployment of applications, where a part of the application is deployed at the core in order to provide more reliability, while another part is deployed on the edge in order to benefit from the low latency.

A great diversity of applications can benefit from *hybrid* core/edge deployments [152, 58]. We focus on collaborative edition environments, web-based applications that allow geographically distributed users to concurrently edit a document. Examples include SaaS applications such as Google Documents, Microsoft Word online, Nuclino and ShareLatex. User-perceived latencies are very important for these applications. Low delays between the action of a user and the visibility of its result by other users reduce the risk of conflicts and improve the global user experience.

While edge servers have been used to improve latencies by offloading localized computation or caching static content [165], the impact of core/edge deployments for dynamic interactive applications, such as collaborative environments, is still relatively unclear [32], in particular for *legacy*, off-the-shelf applications.

The ability of a legacy collaborative application to be deployed over both core and edge resources strongly depends on its software architecture. Monolithic applications linking to a unique database are not a good fit for hybrid deployments, for the same reasons they are not easily scaled horizontally in a single datacenter. A hybrid core/edge deployment requires instead the application to have a modular architecture, in order to be able to move or clone some of its constituents from core to edge resources, based on performance, reliability and durability considerations. The design principle of *microservices* [128] has gained a strong momentum for building large web applications in cloud environments. Under this model, the application is split in a collection of independent single-purpose services whose implementation is independent from that of other microservices. Each microservice may use a different solution for managing its state and be independently scaled horizontally. Interaction between services typically happens through resource-oriented HTTP APIs following REST principles [111].

Microservices-based applications naturally are good candidates for hybrid core/edge deployments. For availability reasons, microservices that handle durable, long-term information should remain in the core, while other microservices can be delegated or cloned at the edge. Deciding which microservice(s) should, or should not, be deployed at the edge is not simple. User-perceived latencies are typically a result of multiple interactions between microservices, and delays between edge and core resources may accumulate and result in poorer performance than in the initial core deployment. Furthermore, while stateless microservices are easier to clone to the edge, stateful microservices require special care, as their state may need to be kept consistent across multiple copies, which can yield more costs than benefits.

In this chapter, we report on our experience of porting a legacy collaborative application for a hybrid core/edge deployment, and present the lessons learnt in the process. Our target application is ShareLatex, an open-source collaborative editing tool for LaTeX documents. The goal is to assess whether porting such an existing complex application to a hybrid deployment is feasible without changing its code, and to identify requirements for overlays which facilitate this deployment.

Our contributions are the following. We discuss criteria for edge deployment of microservices, including aspects linked to state management and consistency, and apply these criteria to the ShareLatex application (Section 5.2). We describe the mechanisms that support this hybrid deployment with no modification to the application source code (Section 5.3). We detail our benchmarking toolset featuring simulated behaviors of users collectively editing LaTeX documents. We experimentally demonstrate the performance improvement in terms of user-perceived latencies for some operations on joint documents and the tradeoffs that exist for other operations due to the hybrid, multi-site resulting deployment (Section 5.4). Finally, we draw some conclusions on how edge overlays can be used in order for this application to take further advantage of the edge (Section 5.6).

5.2 Background

We review microservices principles and define categories of stateful microservices based on their ability to be replicated at the edge. We then describe our use-case application, ShareLatex.

5.2.1 Microservices

Microservices [128] are an evolution of service-oriented architectures, and are currently very popular for building large-scale cloud-based applications. They favor the use of a collection of small and feature-focused software entities over software monoliths. Microservices are advantageous for agile software development, as each service can be implemented, tested, and evolved independently. Interaction between microservices typically happen through APIs following the resource-centric REST over HTTP [111].

While microservices principles were not designed specifically for edge deployments, they offer a number of advantages for this purpose. Due to the statelessness of REST, interactions between services support dynamic relocation. This feature is already leveraged for elastic scaling. In order to support hybrid core/edge deployments, it is desirable to support the creation of *replicas* of services to one (or several) of the edge sites. This ability primarily depends on whether the service is stateful (i.e. it maintains state across different calls) or stateless (i.e. it may maintain temporary state for an ongoing request, but no state is kept between requests). Stateless services are easily replicable. Replicated stateful services are however concerned with consistency, as simply creating a clone of their existing database to another site may lead to diverging and irreconcilable states. Fortunately, we can observe that in many cases replicating or splitting a stateful service and its database to the edge may only have a limited impact on application

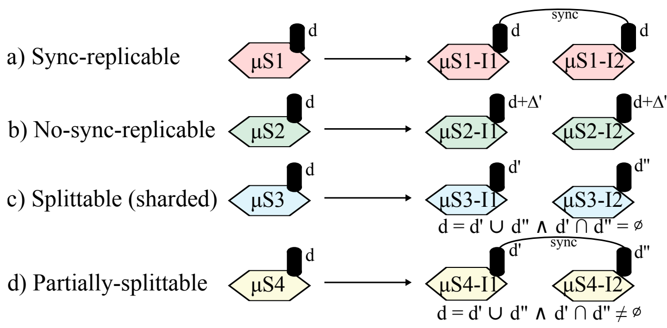


Figure 5.1: Four categories of microservices based on the impact of their replication on application correctness.

correctness. Identifying such services in a legacy application is actually key to deciding which can or cannot benefit from a deployment to the edge. We define four cases, illustrated by Figure 5.1.

• *Sync-replicable* services require the full current state to be available on all instances of the service in order to maintain a correct application behavior. They also require this state to be kept strongly consistent across replicas. Replicating such services generally leads to poor performance/cost tradeoffs, unless they have read-mostly accesses and are able to employ a strong consistency model that allows local reads, such as sequential consistency.

• *No-sync-replicable* services require a copy of the full service state for a new (edge) replica, but can preserve application behavior without enforcing strong consistency with the original (core) replica. The cost for replicating such services therefore only depends on the initial cost of copying the state from the core to the edge.

• *Splittable* services can provide semantically-equivalent service with only a *subset* of the original full service state. As partial states (shards) are disjoint between services instances, there is no need for consistency enforcement. The cost of replication in this case is the cost of splitting and outsourcing one of the shards to the edge.

• *Partially-splittable* services are a hybrid between sync-replicable and splittable services where only a *part of the state* has to be shared and maintained strongly consistent between replicas, while the other part can be split or sharded. Replicating these services to the edge without falling back to the sync-replicable mode is however difficult. It typically requires modifying their implementation to redirect accesses to shared and disjoint states to different databases, or using geo-replicated databases that can leverage the partial split information [66, 96].

The *no-sync-replicable*, *splittable*, and *partially-splittable* assume a preferential attachment from users to a particular site, when accessing the service for a specific *resource*. It is necessary to deterministically redirect calls to this specific site. When there is no modification to the application code, this redirection has to rely on externally-visible information. This is often available in the form of the URI resource name in HTTP REST calls to microservices. This strengthens their potential for edge placement.

5.2.2 Use case application: ShareLatex

Our goal is to study the hybrid deployment of a legacy collaborative editing application, with no modification to its source code. We target an application based on microservices. We select ShareLatex, a collaborative tool popular in research and teaching communities. It enables concurrent real-time modifications by various users on the same LaTeX document.

Collaborators in ShareLatex are likely to come from the same location. Typical examples could be a student and her supervisors, or researchers across local institutions working on a joint paper. Responsiveness is an important aspect for this application, as oftentimes editors can work

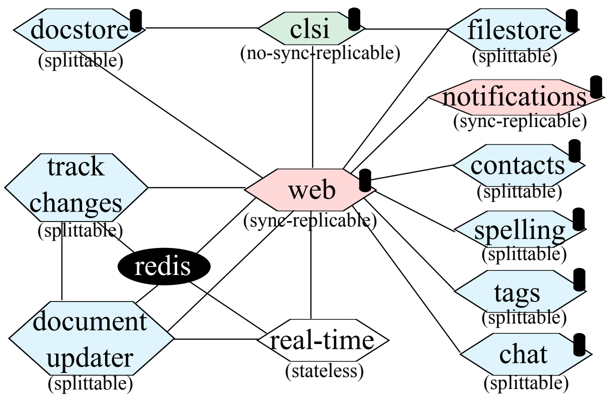


Figure 5.2: ShareLatex microservices and their replication categories as defined in Figure 5.1.

collaboratively on the same parts of the document and need to get updates and notifications from other users in near-real-time.

ShareLatex features 12 microservices and 2 databases, listed in Table 5.1. Their interactions are shown in Figure 5.2. We observe that the interaction is roughly centralized: the *web* service acts as hub for all other services and as the interface to the user (therefore playing the role of an API gateway). We also see that the *web*, *track-changes*, *document-updater* and *real-time* services are all interconnected through a *Redis* database, which is in charge of storing the real-time state of a document under edition (list of modifications, cursor positions, etc). This state is saved periodically in a long-term storage for documents with a call to the *docstore* service.

We should note that microservices principles are best practices in nature. They are not strictly and consistently enforced across entire applications. This is the case for ShareLatex. Service decoupling and state isolation are key principles in microservices. However, we can observe in this application that the *Redis* database is used to share the state between various other services. This inevitably affects the possibilities we have for replicating and deploying these services to the edge. Our goal is to study the impact of hybrid deployment for an unmodified, legacy application. We choose therefore to work with the application as it stands and leave the evaluation of "pure microservices" applications to future work. We note that the lack of availability of such application for benchmarking has been pointed out by other authors before us [4].

Service	Description	Category	Cr.	LS	Fr.	Service	Description	Category	Cr.	LS	Fr.
1. docstore	CRUD ops on tex files	Splittable	✓		✓	8. track-changes	History of changes	Splittable		✓	✓
2. filestore	CRUD ops on binary files	Splittable	✓		✓	9. real-time	Websocket server	Stateless		✓	✓
3. clsi	Compile project	No-sync-replicable			✓	10. notifications	Notifications between users	Sync-replicable			
4. contacts	Manage contacts	Splittable				11. document-updater	Maintain consistent document state	Splittable		✓	✓
5. spelling	Spell checking	Splittable			✓	12. web	User interface and service hub	Sync-replicable	✓	✓	✓
6. chat	Chat service	Splittable			✓	Redis (db)	DB (Pub/Sub) for dynamic data				
7. tags	Folders, tags	Splittable				MongoDB (db)	DB for internal static data				

Table 5.1: ShareLatex services, their categories and additional attributes: critical(Cr.), latency-sensitive(LS), Frequent(Fr.)

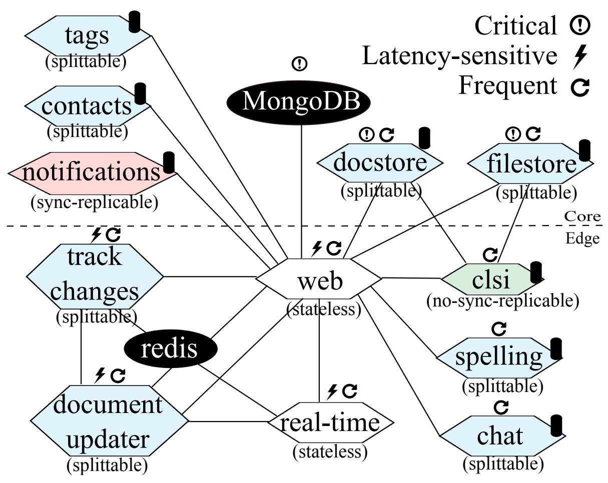


Figure 5.3: Our split and placement. Critical services in the core, latency-sensitive and frequent services at the edge.

5.3 Hybrid core/edge deployment

We seek to determine a *split* of the application, that is a set of replication possibilities for the microservices and a suggested *placement* on core and edge resources. We leverage the mapping into service categories defined in Section 5.2. Categories for all services are listed in Table 5.1. No service in ShareLatex is partially-splittable. Most ShareLatex microservices are splittable services. In particular, we observe that we can split their database either using the project identifier present in the REST calls URIs (services *document-updater*, *track-changes*, *docstore*, *filestore*, *chat*) or by the user identifier available in the same way (services *contacts*, *tags*, *spelling*). Two services, *web* and *notifications* are sync-replicable as their state cannot be replicated without full synchronization. The *clsi* service is no-sync-replicable: replicating it with no further consistency between replicas does not impact correctness. This service handles the compilation of LaTeX sources. For performance, it maintains a cache of the project elements including images and indexes, but losing this state only results in these being fetched again from the *docstore* and *filestore* services.

All services could be potentially replicated from the core to one or more edge sites, but it makes more sense for no-sync-replicable and splittable ones. There are however other non-functional aspects that must be considered for deciding on an appropriate placement. We identify three key aspects: *criticality*, *latency-sensitivity* and *frequency*. Critical microservices are essential for the resiliency and availability of the application. Data loss in this case must be avoided, requiring reliable hardware typically available only in the core, and this is where these services should stay. Services maintaining state but performing periodic checkpoints to another critical service can often be considered non-critical. Latency-sensitive services typically benefit highly from edge replication, especially when they can leverage the locality between users. Similarly, frequently-used services are more likely to impact user experience, and this impact pays off more compared to the cost of enabling their replication.

While service replication categories suggest which services *may* be deployed at the edge, the additional non-functional aspects indicate if they *should* be. Table 5.1 lists this information for all services. We start by discussing a potentially contradicting case for the *web* microservice. This service has a sync-replicable state, meaning it is costly to replicate to the edge(s), but it is also latency-sensitive. Previously, we observed that it is closely coupled to other latency-sensitive services. From the service interaction graph, we can deduce that these services benefit little from the edge deployment if they have to interact constantly with the core. Otherwise, we would have to implement strongly consistent replication which defeats our objective to avoid source code modifications. Our suggested workaround is to decouple the state from the *web* service and host it separately. We deploy the *MongoDB* database of *web* in the core. The now stateless service can be deployed at the edge and benefit from low-latency interactions with other services through the *Redis* service. This is not an optimal solution as database accesses are now significantly less efficient. However, it is the only one which does not require any intervention on the application.

After considering the different service attributes and the workaround, we propose the split shown in Figure 5.3. We keep in the core services that are critical (*docstore*, *filestore*), sync-replicable (*notifications*) and infrequent (*tags*, *contacts*). We deploy on the edge the latency-sensitive services (*web*, *document-updater*, *real-time*, *track-changes*) and the frequent services (*spelling*, *clsi*, *chat*). Note that there remain instances of all services deployed to the edge in the core. These are not used by edge users but are useful for other users who connect directly to the core.

5.3.1 Implementing redirections

When considering the different categories of stateful services, we argued that splittable services can have multiple edge instances with only a partial state of the original core service. However, this comes with an additional challenge. Users of an edge are not necessarily required to access only the partial state of the service instance in that edge. Therefore, in case of a request which can not be satisfied by the local edge instance, the user needs to be redirected to the appropriate instance which has the required state.

In ShareLatex such a scenario happens typically with the state of a project. We hold the instance where the project was first created responsible for the state of the project. For example, if the project was created by a user at an edge, the state of the project will remain in the instances of that edge. As long as this project is shared with users of the same edge, no redirection is required as the local instance has already what is needed. However, if this project is shared with a user who connects to another edge (or core), this user needs to be redirected to the instance where the project was created. Figure 5.4 shows some of these scenarios.

The application itself is oblivious to the fact that the state is split. Therefore, there is no internal mechanism which automatically redirects the user to the right instance. This has to be handled externally. In ShareLatex all user calls are intercepted by a reverse-proxy (nginx) which then redirects them to the local *web* service. Given that this interaction is based on a

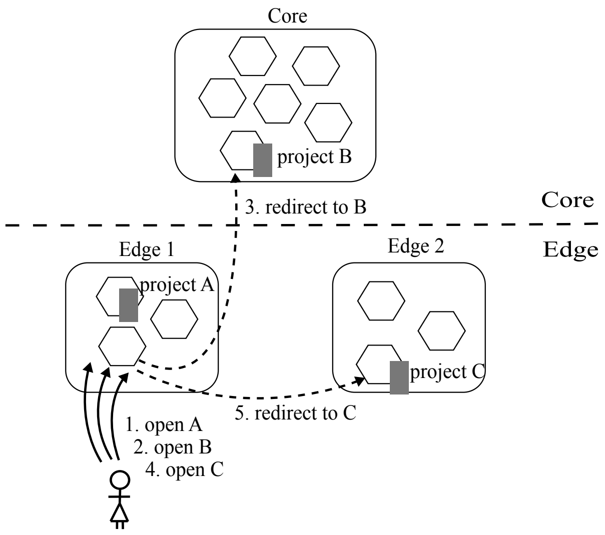


Figure 5.4: Redirection scenarios. Project A needs no redirection, while B and C need redirection to the core and to another edge.

REST API, the URL itself contains information about the object to be accessed (project), such as an identifier. We slightly modify the configuration of this proxy such that, depending on the project identifier, the request can be forwarded to a different *web* instance than the local one.

At the moment, this configuration is done manually and information regarding where each project is stored is also known a priori. Ideally, a mechanism which allows automatic redirection, by keeping track of the location of each project and interacting with the reverse proxy is required for enabling a fully transparent edge deployment. This topic is discussed thoroughly in the next chapter.

5.4 Evaluation

In this section we evaluate the performance of the hybrid deployment of ShareLatex based on our proposed split. We use 3 server-grade nodes from Grid'5000 [63], each with 2 Intel Xeon E5-2630 v3 CPUs and 128 GB of RAM. Our experiments do not focus on scalability or resource saturation. As a result, we use the same machines at low utilization rates for emulating core and edge sites as well as emulated users. We emulate WAN latencies between sites using the `tc` (traffic control) tool. We use a 50 ms roundtrip latency between the core and any edge site, and a 70 ms roundtrip latency between two edge sites. Users are considered as being close to one of the edge sites, to which they connect with a 5 ms roundtrip latency. We package the split ShareLatex application of Figure 5.3 as containers deployed using the Docker CE platform.

Emulating real users. We focus on user-visible application responsiveness. We are interested in delays between actions by one user and the visibility by other users inside the actual web application running in a browser. This latency is a conjunction of multiple factors that cannot be fully modeled by measuring API-level latencies. We emulate a set of users using `Locust` [104], a load testing tool that allows to describe programmatically the behavior of users as a list of actions and their respective weights, which define the occurrence frequencies. Actions are HTTP or WebSocket requests. For example, opening a project requires an HTTP call to the project URL and establishing a Websocket connection. This connection is then used to write content to the project, which is another action which has a high frequency as it is the most common interactive one. `Locust` simulates a number of concurrent users by executing actions and scenarios in separate pseudo-threads (*greenlets*). Our user behavior is similar to the one used by Sieve [159].

To measure end-to-end user-perceived application latencies, we collect timestamps at each action start by some user, and the corresponding events occurring on the application interface of other users (e.g. a cursor position change and its visibility).

5.4.1 Centralized vs. hybrid deployment

We start by comparing perceived latencies between a centralized deployment and a hybrid deployment using a single edge site. We use 5 concurrent emulated users who open and modify the same shared projects. Figure 5.5 shows the setup. Users first share a project fully hosted in the core (project "Core"). This represents our baseline. Measurements for a different set of actions are taken for a period of 10 minutes. Users then switch to working with another project hosted on the edge site (project "Edge") and run the same measurements for another 10 minutes.

We present results for the six following common actions: writing, moving the cursor, spell checking, sending chat messages, compiling the project and displaying the history of modifications. Figure 5.6 shows the distribution of perceived latencies for each of them.

In the first row we observe that for common actions such as writing, cursor update and spell checking, users take advantage of reduced latencies when connecting to the project when hosted on the local edge site. These actions are indeed handled in our split by services (or services

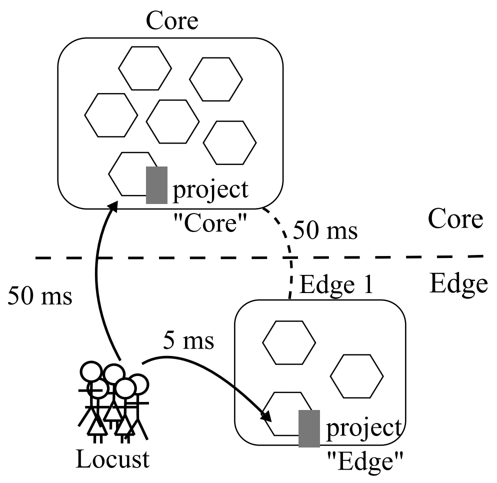


Figure 5.5: Centralized vs. hybrid deployment - exp. setup.

shards) that are all replicated to the edge. The other three actions, present in the second row, are negatively impacted by the hybrid deployment and the hosting of the project on the edge. The main services in charge of these actions are *chat*, *clsi* (for compiling) and *track-changes* (for history). These services have replicas running on the edge site in the hybrid deployment, but their interactions require communication with services instances remaining in the core, which results in additional latencies. For instance, although the document is compiled at the edge, the base document is retrieved first from the *docstore* service, which we kept in the core for availability reasons. In other cases, such as sending a chat message, the added latency comes due to the *web* service which needs to access its remote database to check if the user has the right permission to perform the task and does not cache this information. Our goal is to keep the application unmodified but implementing a fix would be relatively easy.

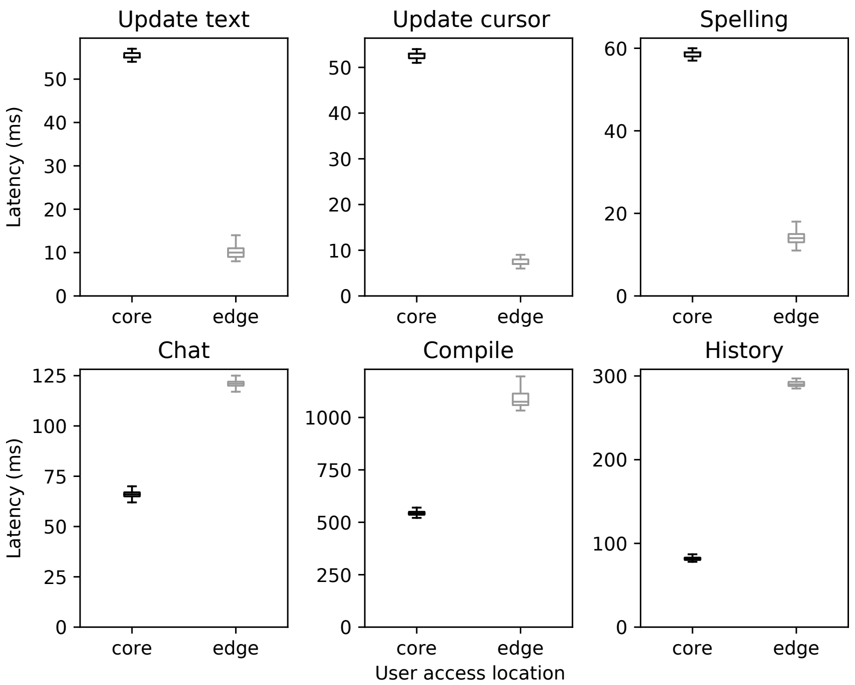


Figure 5.6: Hybrid deployment yields lower latencies for most frequent actions (first row).

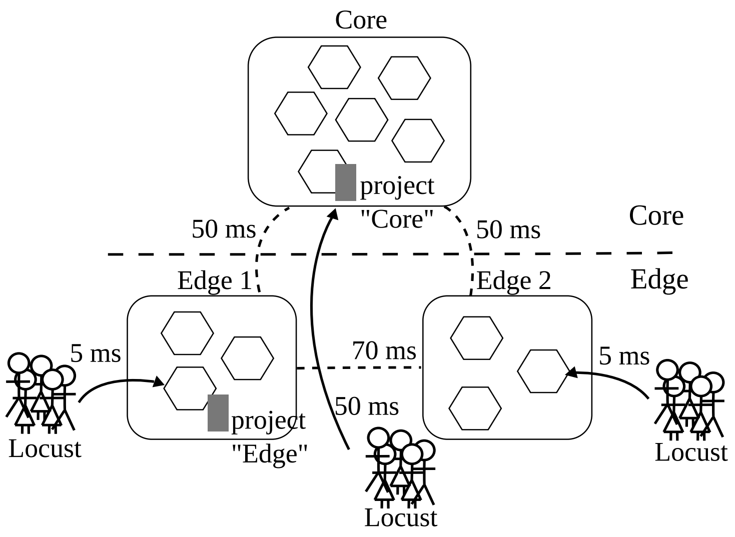


Figure 5.7: Impact of using redirections - exp. setup.

5.4.2 Impact of redirections

In a second experiment, we evaluate the impact of redirections when the project accessed by users is not managed on the site (core or edge) they are connecting to. We consider the scenario with two edge sites, *edge1* and *edge2*, shown by Figure 5.7.

We distinguish three user groups based on the site they connect to: *edge1* users, *edge2* users, and users connecting directly to the core. Two separate projects are hosted on *edge1* and in the core – we refer to them as project "Edge" and project "Core". We emulate 5 users at each location who simultaneously access one project at a time. Therefore, we alternate between the 3 groups and 2 projects for 6 possible configurations.

Figure 5.8 shows the distributions of user-perceived latencies for the six operations we used previously, and for the six considered configurations.

Perceived latencies for updating the text or the cursor position for the "Core" project are similar for all access locations, with a slight increase for users connecting through one of the edge sites. This small increase highlights the low cost of the unique redirection implemented by `nginx`. For the "Edge" project hosted on *edge1*, latencies for these two operations are reduced for local accesses (when users connect to the *edge1* site) while for accesses through the other sites they increase, mostly due to the addition of network latencies between sites, as the redirection operation itself remains of low cost. Observed latencies are in line with the emulated network configuration: 50 ms to the core and 50 ms more from the core to *edge1* in the first case, or 5 ms to *edge2* and 70 ms from *edge2* to *edge1* in the second case. Interestingly, we can observe that the *spelling* operation benefits from locality for users connected to both *edge1* and *edge2*. The reason is that the *spelling* service does not depend on a particular project instance or identifier: there is no need to redirect the call to a particular site if an instance of the service is deployed locally.

The second row of Figure 5.8 presents latencies distributions for operations that always involve interactions with services instances in the core. The cost of redirecting some of the calls to services in the core is expected to have a negative impact when accessing a project located in the edge, as we have shown in our previous experiment. We observe indeed that these 3 actions on the "Core" project have a lower latency than the same actions on the "Edge" project. Furthermore, for these three actions applied to the "Edge" project, we do not observe the better performance for users connecting to *edge2* compared to those connecting to the core, that we could observe for the actions of the top row. The reason for this is that the additional latency of redirection paid by users connecting to the core is evened out by the use of calls to local core

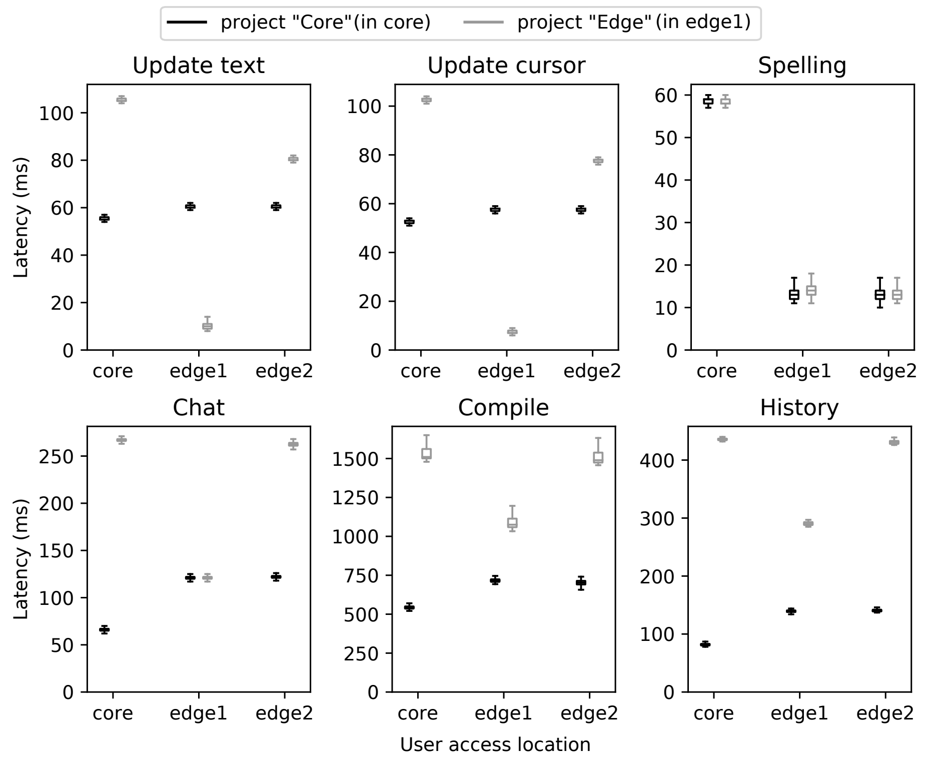


Figure 5.8: Impact of redirection and user access points.

services that follow. Users connecting to *edge1* have better latencies for the “Edge” project as they avoid the price of redirection for the initial call to the service, but still do not perform as well as when the project is hosted directly in the core. One exception is the *chat* service which yields the same delays for users connecting to *edge1* despite the location of the project. This occurs because the cost of redirection happens to be the same as the cost of core/edge crossings for this service.

In general, we observe from our experiments that the hybrid core/edge deployment results in a compromise between the gain of performance of operations that involve services that can be replicated (or split) to the edge site, and the loss of performance that is observed for operations requiring interaction with services staying in the core.

5.5 Comparison to related work

The use of microservice-based infrastructures for providing core/edge integration has been explored by Vilari et al. [163]. Our work targets the evaluation of the hybrid core/edge deployment of a *legacy* application.

A complementary line of work proposes new programming and software engineering models for building new applications in edge/cloud environments. This includes component-based approaches such as Jolie [120], solutions based on eventually-consistent convergent data types (CRDTs) such as LASP [115] or development frameworks for stream processing such as Steel [130]. Additionally, frameworks such as Legion [99] enable existing applications based on Google Drive Real Time API to maintain shared objects (through CRDTs) on the client side in a peer-to-peer way, without significant modifications to the application.

Fesehaye et al. [52] consider a scenario in which an overlay of *cloudlets* (similar to the type of edge resources we consider here) supports applications including file edition. The authors study the impact of document location on user-observed performance. This is complementary to the problem we consider, and we similarly highlight in our evaluation the impact of hosting part of the state of our target application on one edge resource or another.

Clinch et al. [34] evaluate the impact of using cloudlets on the *display appropriation* of mobile applications users, which depends on the responsiveness of the application. The study considers in particular a simple interactive game and evaluates users' perception of its responsiveness for different deployment models. The authors conclude that while there are lower latencies, the user perception is not significantly impacted by the use of close-by cloudlet resources.

Báguena et al. [25] study the responsiveness of mobile apps under core, edge and hybrid core/edge deployments. Their model considers the limited resources and lower latency available at the edge. It proposes a split between services for hybrid placement. It does not consider however, the interaction patterns between these services and the impact they may have on the performance of the application, and it does not provide guidelines for splitting an existing application into services able to benefit from hybrid deployments.

Aderaldo et al. [4] discuss the evaluation of microservices-based applications in general, and note the lack of empirical study on the performance of these applications. Our study aims at filling this gap for the particular case of hybrid core/edge deployments. Sieve [159] also considers ShareLatex as a target application, for the problem of efficiently collecting monitoring information from microservices. The monitoring allowed by Sieve could be the basis for implementing future automated core/edge deployments policies.

5.6 Conclusion

In this chapter we went through the various steps of edge deployment of ShareLatex, a microservice-based legacy application. We argued that the state of a microservice is critical for its replication at the edge. We distinguished four kinds of stateful microservices and explained which of them are more eligible for edge deployment. Furthermore, we considered three additional aspects which are relevant in deciding which of the eligible services should be replicated to the edge. Finally, after taking in consideration both criteria we proposed a split between core and edge.

Through experiments we demonstrated that even without modifying an application, but just its configuration, it is still possible to benefit form a core/edge deployment. This benefit comes mostly from use cases which are handled solely at the edge. Use cases which require interactions between core and edge services tend to be less efficient in this configuration. The overall benefit depends on the relevance of each these use cases on the user experience.

Finally, in this chapter we also highlighted the need for keeping track of the split instances where the projects are located and handle the automatic redirection to them. In the next chapter, we dive into details on this particular topic. We generalize the concept of projects so that it applies to any information which can be split in different instances, and treat the problem of instance localization and redirection as a service discovery problem.

Chapter 6

A locality-aware object-driven service discovery mechanism

6.1 Introduction

In the previous chapter we discussed about how service oriented architectures (SOA) based on microservices enable edge deployment of applications. Microservices were distributed between core and different edges but the details of how they were able to locate each other in the network were not given. Due to their dynamic nature, microservices can not know a priori the network addresses of all the other microservices they need to interact with. During its lifetime a service might need to interact with different instances of the same service which have different dynamically assigned network addresses. Therefore, a dynamic service discovery mechanism needs to be in place.

Service discovery is indeed an integral part of the original SOA design. Despite the differences in API and protocol complexity, the main concept remains the same for microservices. Such mechanism relies on maintaining a distributed registry of running services and details about them, such as their identifier (service name), network address and port for each instance of a service. Service discovery consists on locating the best fit instance of a service by only specifying the service name.

Determining the best fit instance depends on various criteria which can be functional or non functional. Functional criteria are critical for delivering the expected behaviour of the application. Among different instances of the same service only some (or one) of them might be eligible to serve a particular request. In that case, service discovery must select an instance which can provide the desired functionality. On the other hand, non functional requirements concern the quality of service. In case more instances can serve a request, the discovery mechanism should propose the instance which delivers a better response time. In the context of an edge environment response time is closely linked to locality awareness. Selecting the closest instance (in terms of network latency) from the request origin is crucial.

In the previous chapter we defined a taxonomy of microservices based on their state and indicated which of them are suitable to be deployed on the edge. Such services included stateless services and two kinds of stateful services, namely: non-sync-replicable and splittable (sharded). Stateless and non-sync-replicable services are replicated and each replica is able to serve any request. In that case service discovery focuses in non functional requirements, such as locating the closest instance. However, for splittable services a single instance is only responsible for one part of data. Therefore, service discovery must locate the instance which is responsible for the data needed to serve a request. We refer to the collection of data specific to a request as an *object*. In case of splittable services, one instance can handle requests for a set of objects which is different from the set of objects handled by another instance. In order to identify the instance responsible for an object, the service discovery mechanism needs to keep track of which objects are handled by each service instance.

We assume the distribution of objects in instances is done externally from the application. That is, objects are not sharded instances due to some internal splitting mechanism, but they are rather sharded naturally by the way they are accessed. In other words, an object is stored in a particular instance because the client issued the request for the creation of that object to that instance. This way the distribution of objects in different instances should reflect the distribution of the clients accessing those objects.

An object can be manipulated by different services at the same time. We refer to services manipulating the same object as the *object service group*. The location of a particular object can determine whether the discovery mechanism would favor one set of instances of its service group over another. This is an important concept because for object related requests, service discovery should not use the origin of the request as the point of reference to allocate the closest instances, but it rather uses the location of the object.

Object location with respect to its client is critical for the performance of the application. We try to store the object initially in the instance closest to the client who created it. However, other clients of the same object (or even the creator) can be located in various other locations. The object can not be bound to the location it was created if its users move away or are distant from it. Ideally, the object should migrate to the instances close to its users.

In this chapter we describe the transformation of the Koala overlay introduced in Chapter 4 into a service discovery prototype. Additionally to traditional service discovery mechanisms, our prototype not only locates the closest service instance, but also the instance responsible for particular object. Most importantly, it is also able to handle object migration based on the location of its users. We integrate our prototype with the edgified ShareLatex application discussed in Chapter 5 and demonstrate its efficiency based on real-life user scenarios.

This chapter is organised as follows: Section 6.2 revisits the concepts of services and objects used in service discovery. Sections 6.3 and 6.4 introduce the details of our prototype and how we integrate it with ShareLatex. Sections 6.5 describes the evaluation of this prototype, while Section 6.6 discusses its adoption by other edge applications. Section 6.7 puts our prototype in context with other existing tools, and Section 6.8 concludes.

6.2 Services and objects

Services are central to the SOA architecture and an integral part of any traditional service discovery mechanism. Given an identifier, or the name of a service, such mechanism should identify the appropriate instance of that service where the request should be forwarded. Such a task is relatively straightforward in case all the instances of the service are equally eligible to serve the request. However, as we saw in the previous chapter, this is not always the case. For some services a single instance is responsible for only a part of the overall data, and therefore it can only support requests concerning that data. We called them split or sharded services.

In the context of edge computing, the way data is sharded between instances should take another factor into consideration, data proximity to its users. One of the prerequisites for applications to benefit from edge deployment is that users in a particular location access and manipulate the same data. Consequently, an instance of a sharded service in one location should consist of data which is relevant for the users of that particular location.

This link between data within each instance and the location of its users introduces three main challenges. First of all, we need to make sure that data is initially stored in the closest instance to its creator. Secondly, we need to be able to locate the instance with the right data, regardless of the location of the requesting user. Finally, we need to deal with the mobility of users and make sure that at each moment in time data is stored where the users interested in that data are.

We address these challenges by extending the traditional service discovery mechanism to deal with the internal state of services. Our goal is to influence the choice of instances where data is initially stored, use the same mechanism as for the services to locate data, and be able to suggest instances where data should be relocated in case of user mobility. We do this by introducing a new concept which correlates data and user locality, we call this an *object.* In the following, we examine two aspects of what we refer to as an object: object as data managed by services, and object as a location reference managed by the service discovery.

6.2.1 Object as data

The first aspect is closely linked to the internal data model of a service or a group of services. In order to be scalable, highly available, and self-contained, services often adopt a non-SQL data model. Common cases include key-value stores or document stores. In this model, data is designed according to user requests such that, in the best case, one request accesses one key, or document. It is common for a document to contain the key of another document, and therefore the request can access a chain of documents referenced by each other. However, all the accessed documents can be tracked to an initial key. We refer to the list of documents accessed by a single request as *object as data* and the initial key as the *object identifier.*

In a standard RESTful call which is specific to an object, the object identifier is always part of a HTTP request. It can either be part of the URL or it can be part of the body of the request. A service discovery mechanism needs to know how to identify the object identifier within the request in order to support object localization. This is discussed in more details in Section 6.4

An object can be distributed across different services, but its identifier remains the same in all of them. We call the set of services working on the same object the *service group* of that object. Consequently, the specific service instances which handle the same object at one point in time are called its *instance group.* For instance, Figure 6.1 shows a traffic monitoring application with two services handling different requests on an object, in this case a road, with identifier *"123".*

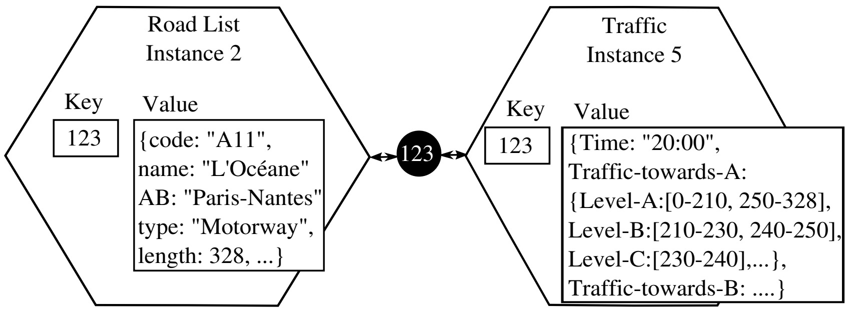


Figure 6.1: Object (road) "123" service group consists of "Road list" and "Traffic" services and its instance group consists of instance 2 and 5 of those services, respectively.

Note that an instance group of an object does not necessarily reside in the same server, but can be distributed among different locations. Our goal is to search for the instance group of an object starting from to the location of its users. In order to define the users location and how that is used to find the right instance group, we need to explain the second aspect of our definition of *object.*

6.2.2 Object as a location reference

In order to search for the closest instance group for a particular object, we need to start from one specific location which represents the location of the users of that object. Determining this location is not always straightforward. In the context of edge computing, users of an object are supposed to be concentrated in one edge location. While that is generally the case, distribution of users between different edge locations is to be expected, especially considering their mobility. Therefore, we need to map different user locations into a representative single one.

A potential candidate location is the one which minimizes the latency with respect to all the user locations. Assuming we can represent edge locations as points in a system of coordinates where the distance between two points expresses the latency between them, finding a point which minimizes the latency from a set of other points means to calculate their geometric center. Nevertheless, by using the geometric center to represent the location of users, we assume that all users are equally important. In reality, some users can be more active, or they can perform different operations which are more relevant to have lower latency than others. If we take into consideration the relevance (weight) of a user, identifying the users location becomes a *center of mass* problem. What defines the relevance of a user is application dependent, and therefore outside of our scope. However, as an illustrating example in this study we use the user activeness. In other words, the more requests per unit of time users make, the higher their relevance.

Once the center of mass of user requests is calculated, we can use the closest edge location to that point as the representative of the locations of the users of an object. Ideally, the whole instance group of an object is deployed in this location, but that is not necessarily the case. Therefore, the service discovery can use this location as a point of reference from where to start looking for the closest instance group.

The association of the object identifier with the representative location of that object's users is what we refer to as *object as a location reference* in the service discovery mechanism.

This is very similar to how services are treated in service discovery. They also run in a particular location and have an identifier (name). The only difference is that while the location of a service indicates where it runs, the one of an object does not represent the location of the object data but the one of its users. Therefore, it is potentially much more dynamic.

By arbitrarily changing the location of an object reference in the service discovery mechanism, we can change the instances which will receive the requests regarding that object. For example, in Figure 6.2 the service group of object O is composed of services S1, S2 and S3. If the location of the object is at *edge1* the instance group consists of S1-I1, S2-I1 and S3-I1. If we move this reference to *edge2* the instance group for this object would change to S1-I2, S2-I1 and S3-I2. Similarly, moving to *edge3* would result in the instance group S1-I3, S2-I3 and S3-I2. This happens because we locate instances which are closer to the object location. If we constantly monitor the location of the users of an object, we can continuously adjust its location to the one of its users. Hence, we make sure that we are always using the closest instance group to the users.

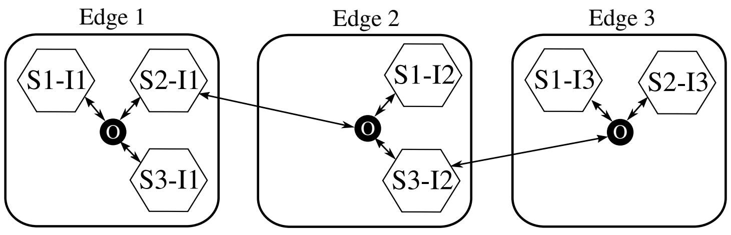


Figure 6.2: Moving object O in one of the three edge locations redirects the requests to different instances of services S1, S2, S3.

This way of dynamically changing the instance group of an object has a limitation which was anticipated earlier. Even though requests are redirected to new instance groups, that does not mean that data related to this object are moving to these new instances. The new instance group might not be able to serve the requests. In order to overcome this discrepancy between object as data and object as location we need an interface between the application and the service discovery such that the service discovery can indicate to the application where to migrate object as data. We return to this topic in section 6.6. For now we assume such a mechanism is in place.

6.3 Towards a Koala prototype

In Chapter 4 we introduced the Koala overlay and discussed its benefits in an edge environment. We mainly focused on its architecture and routing protocol but did not give any details of how it can be employed in a real world scenario. In this section we dive into more details regarding the use of Koala DHT as a building block for a service discovery mechanism.

In the previous chapter we discussed the edgification of a microservice-based application. We talked about a core location and different edge locations, each of which is composed by various servers. We considered that each server is virtualized using some container technology such as Docker and that each microservice runs as a separate container. In order to use Koala as a service discovery mechanism for this application we need to adapt it to a prototype compatible with this infrastructure and define how it interacts with the application.

6.3.1 Architecture overview

The first step for converting Koala into a prototype was mapping each logical node in the simulator to a stand alone web server which can interact using a RESTful interface. For this prototype we opted for a lightweight server-side run-time environment such as *Node.js*. Due to its simplicity, such choice is common in microservice-based applications, including ShareLatex.

These Node.js Koala servers are packaged into containers and run on every host along with other microservice containers which constitute the application. Koala servers must be reachable from outside their host. In this way, they interact with each other and create the ring-based Koala overlay and DHT (see Figure 6.3). However, two relevant components for initializing the overlay and storing service information in our DHT are the *boot node* and the *registrator*.

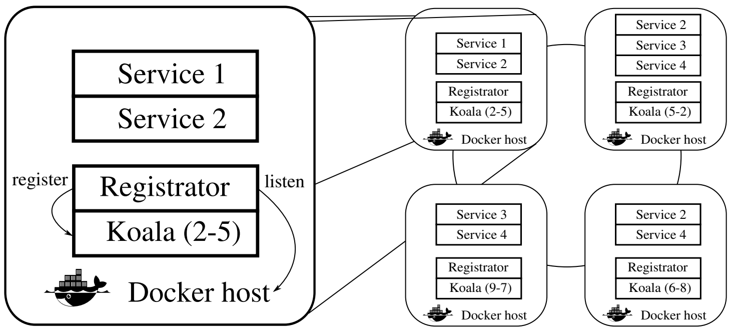


Figure 6.3: Overall architecture

6.3.1.1 Boot node

As described in Chapter 4 the creation of the overlay starts with an initial boot node whose address is known to other nodes. In our setting, the boot node is yet another Node.js web server which runs along a standard Koala server within the same container. In fact, each Koala container can be a boot node just by enabling a configuration flag. By default, only the Koala container which runs in the core has the boot node flag enabled as the core address is known by all other nodes.

6.3.1.2 Registrator

As mentioned earlier Koala servers provide a RESTful API and two of its main functionalities are to register and deregister services. However, often applications are not aware of the existence of a service registry, therefore there is no explicit registration from the application itself. A more common pattern for registering services is by using a third-party registrator.

In our setup we adopted the *registrator* tool by GliderLabs [90]. This tool runs as a container on each host along with Koala and the other services. Its particularity is that it is provided with access to the Docker socket of the host. Therefore, whenever a container with a specific tag starts or stops on its host, this tool can notify any external service registry by providing it with data about the container. These data include the container (service) name, network address, ports, etc. By default, the registrator supports interaction with traditional service registries such as: Consul [71], etcd [37] and SkyDNS [149]. We extended it to support registration and deregistration of services in Koala. The Koala registry is described in the following.

6.3.2 Internal state

So far we have seen the overall architecture and infrastructure on which our prototype operates without depicting its internal mechanism. We also saw how information about services was introduced to Koala. In this section we dive into details about how the state of services and objects is stored in Koala.

6.3.2.1 Services

Before discussing about service data in Koala, we briefly recall the way data is generally stored in a distributed hash table and then instantiate that to our particular service discovery. As in many DHTs, each Koala node has an identifier (or a key) from a given circular identifier space. Each node is responsible for keys between its own and that of the next Koala node in the ring. Data is mapped into the identifier space by means of a hash function which takes as input an unique attribute of these data.

As a hash function we use djb2 [45] due to its fair performance over value distribution ratio. The name of the service is provided as an input to this function. Since the service name is common for all its instances, data about all these instances are stored in the same Koala node. In this case, we say that that node is *responsible* for that service.

In addition to the services for which it is responsible, a Koala node stores also services which are running in the same host with it. We call these *local services*. As we mentioned earlier, the registrator notifies Koala about services running in the same host. These services are stored as local, and their responsible nodes are notified about their location. We will give further details about service registration in section 6.3.3.

Service name	Instance ID	Location	Responsability	IP	Port
Service 1	Instance 1	local	responsible	x.x.25.1	3001
Service 2	Instance 1	local	responsible@6-8	x.x.25.2	3002
Service 3	Instance 1	remote@9-7	responsible	-	-
Service 3	Instance 2	remote@5-2	responsible	-	-

Figure 6.4: Service data in Koala

Figure 6.4 illustrates the data we store about each service instance by taking as example Koala node 2-5 from Figure 6.3. A Koala node stores data about services which either are local (Service 1 and 2) or those for which it is responsible and which might not be local (Service 3). For local services the IP and port are stored, whereas for remote ones we just store the Koala node local to them in the routing table, as we discuss shortly.

6.3.2.2 Objects

Earlier we mentioned that objects are treated very similarly to services in Koala. The object identifier is hashed in order to determine its responsible node, and its location is used to determine if it is local. Therefore, a Koala node keeps data about objects for which it is responsible and those which are local (Figure 6.5).

Object identifier	Location	Responsability	Instance group
Object 1	local	responsible	Service 1 - Instance 1 - local, Service 3 - Instance 2 - remote@5-2
Object 2	local	responsible@6-8	Service 3 - Instance 1 - remote@9-7
Object 3	remote@5-2	responsible	- -

Figure 6.5: Object data in Koala

One difference in objects is that we keep also information about the instance group where the object as data is stored. Upon creation of an object, a certain instance group is initially selected to store its data. As long as the location of the users of that object has not changed, we need to keep track of where data is stored in order to avoid searching for them on every request. The way objects and their instance groups are registered and maintained, as for the services, is described in further details in section 6.3.3.

6.3.2.3 Routing table

The kind of data we keep in the routing table of a Koala node was discussed extensively in Chapter 4. However, in that chapter we did not consider a specific application of the Koala overlay, and therefore certain details of how some of the entries were populated and used were omitted. In this section we provide further details about entries in the routing table by considering the context of a service discovery tool.

We briefly remind that the routing table consisted of five kinds of entries: neighbors, long links, proximity links, random links, and application links. While all others are generic to any adoption of Koala, the choice of application links depends on the specific application, which in our case is the service discovery. Application links are entries in the routing table which optimize the performance of the application build on top of Koala. As mentioned earlier, a node responsible for a service should know about all the locations of the instances of a service, i.e. the Koala nodes local to them. Furthermore, if a service is running locally to a node, this node should also know about the responsible of that service. All these links are stored as application links.

Network coordinates. In order to provide locality-aware service localization, Koala keeps track of the node locations. This is done by maintaining Vivaldi coordinates [41] for each entry in the routing table. When requesting a service a node specifies also its own coordinates. As the requests reaches the responsible of the service, this last one calculates the distance to all the possible instances and proposes the one with the smallest distance. Figure 6.6 shows a part of the routing table of node 2-5, which we considered earlier too, including the Vivaldi coordinates.

	Node ID	IP	RTT	Coordinates
...		...		
Application links	5-2	x.x.5.2	25	(x', y', z')
	6-8	x.x.6.8	43	(x'', y'', z'')
	9-7	x.x.9.7	62	(x''', y''', z''')

Figure 6.6: Application links for node 2-5 with Vivaldi coordinates.

In our prototype we use the classical Vivaldi algorithm without further optimizations. Following the philosophy of Koala, coordinates are calculated in a lazy way. This means that we do not create traffic for transmitting the coordinates, but they are piggybacked at each service request or look up. At each request we use the *pcap* library [135] for calculating the RTT and update the coordinates accordingly.

6.3.3 Protocol

While in the previous section we presented a snapshot of the state within Koala, here we focus on the interactions which manipulate that state. More specifically, we talk about interfacing between clients and Koala, the way services and objects are registered, and we dive in more details of the internal mechanisms of Koala, that is, the interaction between Koala nodes in order deliver its services.

6.3.3.1 Interface to application services

Traditional service discovery mechanisms such as Consul or Etcd are in general used as a key-value store. A service is represented by a simple entry with its name as the key and the network address and other details as value. Commonly, a service based application which uses service discovery does a lookup for a particular key and then, once it finds it, it makes the requests to the retrieved service address and port. This is a fairly simple approach, but it requires the application to have a built-in mechanism for interacting with the service discovery. In Koala we chose a different approach. Koala acts as a proxy between the services of the client application. In other words, each service instead of contacting other services directly, sends the request to the local Koala instance, which is responsible for proxying the request to the final destination. This is inline with the philosophy of Koala which uses application traffic to maintain the overlay: By doing this redirection, all the application traffic goes through Koala.

Services: Configuring the application to redirect generic service requests is in general straightforward. If services are statically configured, the URL of the server service is written in a configuration file of the client service. Therefore, by changing that configuration file we can redirect the request of that service to the local Koala which should run in a known port. For instance, if service A is configured to send requests to service B, it will have in its configuration file the URL of service B such as: `http://service_B_host:service_B_port/`. In order to make it send requests through Koala that URL is changed to: `http://koala_host:koala_port/api/get/service_B/`. When Koala receives such a request, it parses the full URL (including the path added by the application), realizes it is for service B and forwards it to the closest instance of that service using the original URL (see Figure 6.7).

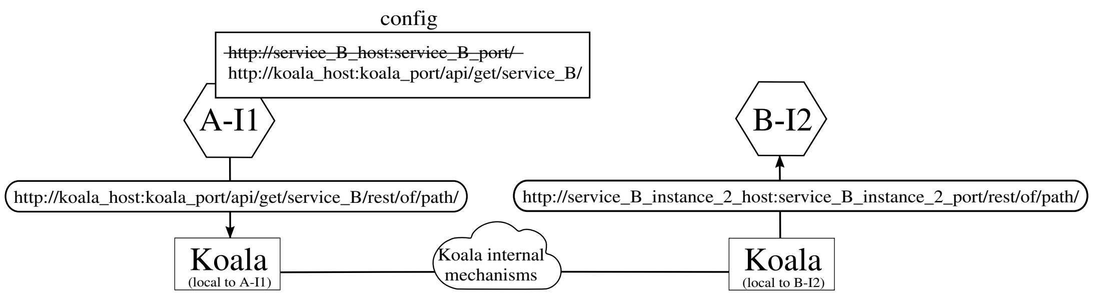

config
~~http://service_B_host:service_B_port/~~
http://koala_host:koala_port/api/get/service_B/
A-I1
B-I2
http://koala_host:koala_port/api/get/service_B/rest/of/path/
http://service_B_instance_2_host:service_B_instance_2_port/rest/of/path/
Koala
(local to A-I1)
Koala internal mechanisms
Koala
(local to B-I2)

Figure 6.7: Configuring services to interact with Koala

Objects: Configuring the redirection for the objects is more complicated and requires an additional actor: a reverse proxy within the service. The reverse proxy is used as a gateway to the service in order to monitor all the incoming or outgoing requests. Its task is to detect object-related requests and redirect them to Koala by indicating where the object identifier is within the request. Therefore, object redirection is done by re-configuring the reverse proxy. For example, in Figure 6.8 the user makes a request for an object with identifier "123" to an instance of service A ①, the URL might look as following: `http://service_A_host:service_A_port/path/123/rest/of/path/`. When the reverse proxy in front of service A receives this request, it will redirect it to Koala by slightly modifying the URL such that it indicates which part of the URL contains the object identifier ②. The URL of the redirected request might then look as following: `http://koala_host:koala_port/api/get/service_A/object/123/callback/cb/path/123/rest/`. In this case, we use the keyword "object" in the path to specify that the next element ("123") is the object identifier and the keyword "callback" to specify that the next element ("cb") is a tag to be used from Koala to instruct the reverse-proxy to forward the request to the service instance. Koala will receive this request and find the instance of service A which actually contains the object "123" ③. Subsequently, it forwards the request to this instance by adding the tag "cb", which instructs the reverse-proxy of the retrieved instance to let the request go through to the instance and not come back to Koala ④.

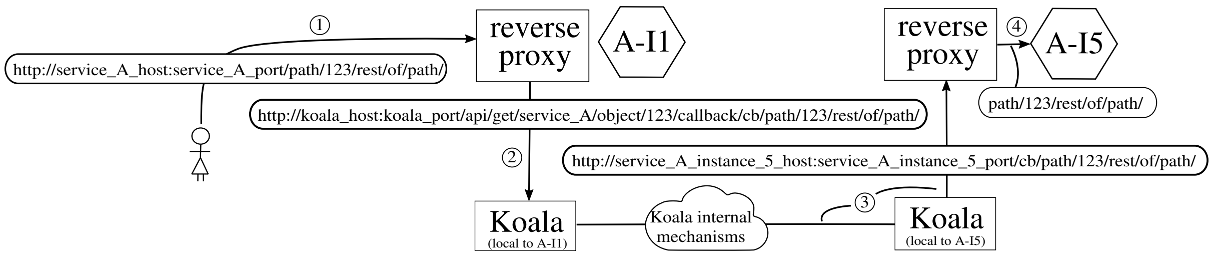

①
reverse proxy
A-I1
http://service_A_host:service_A_port/path/123/rest/of/path/
http://koala_host:koala_port/api/get/service_A/object/123/callback/cb/path/123/rest/of/path/
②
Koala (local to A-I1)
Koala internal mechanisms
③
Koala (local to A-I5)
http://service_A_instance_5_host:service_A_instance_5_port/cb/path/123/rest/of/path/
reverse proxy
④
A-I5
path/123/rest/of/path/

Figure 6.8: Configuring object redirection with Koala

6.3.3.2 Registration

Registration of services and objects in Koala is triggered by different actors but it remains relatively similar.

Services: In case of services, we saw earlier that the registration is triggered by the registrator. Koala offers an explicit API which is used by the registrator to contact the local Koala and register the details of local services. During registration, the Koala node computes the hash of the name of the service and if the node itself is not responsible, it looks up for the responsible of that node while providing details of itself. Upon receipt of the message, the responsible stores information about the service instance and adds the sender as an application link.

Objects: For objects, registration is more implicit as they are registered based on user requests and not by a third party. In order to register an object, the object identifier is required. However, a request which creates a new object does not contain its identifier, as the identifier is created as a result of this request. In this case, the application should indicate to Koala that the current request will create an object. Furthermore, in the response it should indicate the identifier of that object too. By knowing that the request creates an object, the Koala node which received this request can track the instances where the request is going to be forwarded and only at the response store the object identifier together with the instance group. This requires some specific interfacing between the application and Koala.

We assume that the Koala node which receives the object creation request is the closest one to the initial user of that object. Therefore, it will store the object reference locally and will look for service instances which are close to itself for storing the object data. As for the services, it will also compute the hash of the object identifier and notify the responsible of that key about the location of the object, if that is not itself.

6.3.3.3 Internal interactions

In this section we examine the interactions between Koala nodes when a user request is issued, as well as its internal mechanisms for optimizing these interactions. More specifically, we focus on the path a request follows from the user to the final instance, caching, and the user location monitoring mechanisms.

a) Request handling. The way generic service requests and object-related requests are treated in Koala are in essence similar, but they slightly differ due to the difference in meaning between a local service and a local object. The instance group of a local object does not necessarily consist of local instances. However, for both kinds of request we distinguish three scenarios depending on the location of the instance of the requested service or the location of the requested object.

Services: The first and simplest scenario is when a Koala node receives a request for a local service. Given that Koala favors always local instances, it will simply forward it to the local instance without any interaction with other nodes.

The second scenario is when the Koala node which receives the request does not have a local instance, but is responsible for the service which is being requested. As a responsible, this node contains a list of all the Koala nodes which have an instance of the requested service local to them. Moreover, it has also their Vivaldi coordinates. Therefore, the node will compute the distance between its own coordinates and those of each node in the list and select the one with the smaller distance to forward the request. Note that the request is first forwarded to the Koala node local to the service instance, which in turn forwards it to the actual instance. Keeping the application traffic between Koala nodes helps with the maintenance of the overlay as additional information is piggybacked.

The third scenario is when the Koala node which receives the request does not have a local instance and is not the responsible of the requested service. In this case, this node will hash the name of the service and initiate a lookup request for the node responsible for that service. Within this request it will also piggyback its own Vivaldi coordinates. Eventually the request will reach the responsible node which will read the coordinates of the sender and propose the instance which is the closest to it in response. Finally, the sender will forward the request to the Koala node which is local to the proposed instance.

Objects: We consider the same three scenarios for object-related requests and assume the object is already registered as described earlier. We further distinguish between a request for a service which is already present in the instance group and one which is not yet there.

In the scenario of a local object request, if the requested service is already registered in the instance group, the request is forwarded to the Koala node local to the registered instance. Otherwise, if the service is not registered, the Koala node will lookup for a service instance which is close to itself. This might follow any of the three scenarios described above for the services. The result of that lookup will be stored in the instance group and the request will be forwarded to that instance.

In the scenario when the requested object is not local but the node receiving the request is responsible for it, this node simply forwards the request to the node local to that object. Note that since objects are unique, there is only one node to which they are local. Once the request arrives to that node, the rest is handled as in the case above.

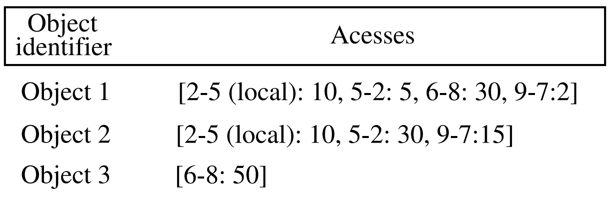
In the third scenario, when neither the receiving node is responsible nor the object is local, the same exact procedure is followed as for the services. The object identifier is hashed, and a lookup for the responsible is initiated. The lookup indicates the only location of the object and the request is then forwarded to the Koala node of that location. In the last two scenarios, when the object is not local, the node which receives the request always piggybacks its own coordinates to the node local to the object. This is important for user tracking described shortly.

b) Caching In the third scenario described above, when the node which receives the request is not responsible or local to the service or object in question, a lookup request is required. This lookup might have a high cost as multiple hops might be required till the responsible is reached. Given that requests for the same service or object are highly probable to reoccur in a short period of time, we cache locally the results of previous lookups in a key-value store, where the key is the name of the service or object and the value is the details of the Koala node local to them.

In order to keep it updated, the local cache is invalidated through two mechanisms. The first one relies on a periodic invalidation after a certain threshold of requests is reached. At that point, the next request re-initiates a lookup which updates the cache. This is especially useful when the cached result is still functional but not the optimal option. The second invalidation happens when the service or object is not anymore available. In this case, the stale destination notifies the requesting node to restart a lookup for locating the new destination, and eventually updating its cache.

c) User tracking and object migration As we see later, we try direct user requests first to the Koala node which is the closest to that user. This means that the coordinates of this node are also the ones which can better represent those of the user. Previously we mentioned that for object-related requests, when the object is not local, the coordinates of the first node are piggybacked together with the request to the node which is local to the object. As a result, a node can know where the requests for all its local objects are coming from.

Similarly to the cache, a node maintains a key-value store of object accesses, where the key is the identifier of the object and the value is an array of accesses. An access consists of the Koala node ID where the request came from and the number of requests as shown in Figure 6.9. Note that the coordinates of those nodes are stored in the routing table.



Object identifier	Acesses
Object 1	[2-5 (local): 10, 5-2: 5, 6-8: 30, 9-7:2]
Object 2	[2-5 (local): 10, 5-2: 30, 9-7:15]
Object 3	[6-8: 50]

Figure 6.9: Object access records of node 2-5.

After the sum of requests for a certain object reaches a threshold, the node evaluates if the object is in the right location by calculating the center of mass of its user locations. The node computes the distance between the center of mass and all the nodes in its routing table, including itself. If a node other than itself results to be closer to the center of mass, then the node initiates an object transfer request to that node which in turn will register the object as local, and notify its responsible for the new location. The last step is not required if the responsible is the sender or the receiver of the object.

6.3.3.4 Load balancing: Koala DNS

One prerequisite of our prototype to work properly is that users should reach first for the service instance running on the closest edge to them. For instance, if the edgified application consists of a microservice-based web application, there is probably a service which runs the front-end where the users connect. This front-end service might have instances in many edge locations where there are users of this application. Therefore, the question is: How can we make the user connect to the closest front-end instance?

This is a common problem which has been tackled by companies which deal with large-scale infrastructures. Details about how this problem is handled in Google and Microsoft are given in [18] and in [26]. Some of their techniques can be adopted also for our edge infrastructure, considering our specific features.

One way to deal with the problem is to use Koala nodes as DNS servers and assign them an anycast address. Koala provides a DNS interface and can be configured as the nameserver for applications running on top of it. The anycast address assures that the request of the user for a name resolution arrives to the closest Koala node. Once there, the request will follow one of the three scenarios described above, in request handling, and will return the closest instance of the front-end.

6.4 Integration with ShareLatex

In this section we focus on the practical aspects of using Koala as a service discovery tool. We use the edgified ShareLatex application we described in the previous chapter as a use case. While previously we talked conceptually about services and objects, here we consider the concrete services and objects in ShareLatex.

6.4.1 Core services

When edgifying ShareLatex we distinguished between the core location and the edge locations. Some services were deployed only in the core, while others had many instances in various edge locations. In Koala we do not essentially distinguish between the core node and the edge nodes as all nodes are treated equally. However, making Koala core-aware can further optimize service localization.

If we have prior knowledge that a service is deployed only in the core and nowhere else, then the discovery is not really needed. Koala allows to tag a node as *core*. By means of the boot node and piggybacking, the information is disseminated so that all the edge nodes are aware of the identity of the core node. Additionally, by a convention for the service we can indicate to a Koala node to treat core service requests differently. For instance, by updating the name of the *docstore* service from "sharelatex-docstore" to "sharelatex-docstore@core", whichever Koala node gets that request will not try to find the responsible for that node but simply forward it to the core.

6.4.2 The ShareLatex object

The main example of an object in ShareLatex is a project. We defined the object as an association of an identifier and a location. Each project has an identifier which is always part of the URL. For instance, the last part of the path of the following URL represents the project identifier: `https://sharelatex.irisa.fr/project/5c5d53ec96d8944976b027b7/`.

A project has also a location which corresponds to the location of its users. Project data are sharded between instances in different edge locations. Ideally, a project-related instance in one location should only contain data about projects whose users are in that location. If the users move, the project data should follow them.

A project has also a service group. As shown in Figure 6.10, several ShareLatex services operate on a project. The *document-updater* keeps the state of the project updated, the *track-changes* maintains the history of changes, the *clsi* compiles the project and *chat* sends messages to the users of the project. All except *clsi* contain data about the project. Therefore, if the project is moved to another location, new instances of these services should be located as explained in Figure 6.2.

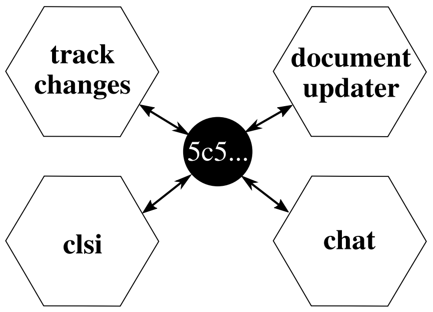


Figure 6.10: Service group of a project.

6.4.3 Interfacing Koala with ShareLatex

Services: Redirecting ShareLatex service traffic to Koala is straightforward. Each service uses a configuration file where all the URLs pointing to other services are set. In addition to this, these URLs can also be set by means of environment variables. For starting multiple ShareLatex services, we use the *docker-compose* tool, which allows to define environment variables for each service. Therefore, pointing all the service URLs to Koala as described in section 6.3.3.1 is a question of modifying URLs in the *docker-compose.yml* file.

Objects: Redirection of object-specific requests is slightly more complicated. In ShareLatex users interact directly only with one service, the *web* service, which acts as a gateway to other services. Earlier we mentioned that for object-related requests, a reverse-proxy in front of the service is required. The *web* service is already equipped with a reverse-proxy which is in this case *ngnix*.

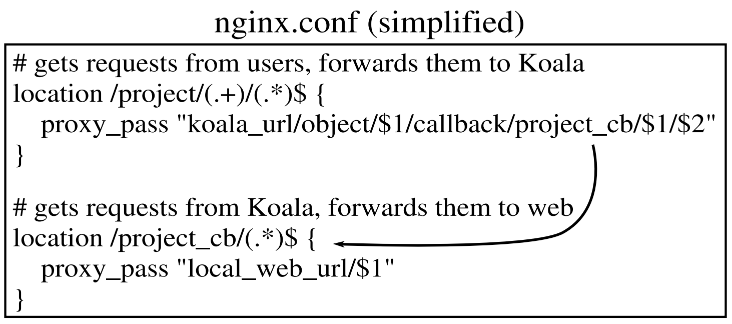
Every request goes first through *nginx* and depending on the requested URL it is either handled by nginx itself or is forwarded to the local *web* service. Nginx is highly configurable by means of *location* directives defined in the *nginx.conf* file. We can specify the kind of URL that needs to be treated differently by means of regular expressions.

We are interested to redirect project-related requests to Koala, by specifying the identifier of the project to Koala. Therefore, we need to identify a regular expression for all these requests. Fortunately, the identifier of the project in ShareLatex is present in the URL for all project-related requests and it comes always after the keyword "/project/", so the definition of regular expression is rather simple (Figure 6.11). We define two main *location* directives. One to redirect object requests to Koala and one which Koala uses to give the request back in case the current *web* service contains the project.

6.4.4 Integration subtleties

In principle Koala is just a network of proxies which redirect the requests to the most appropriate service instance. In theory, it should not intervene in the request or alter it in any way. It should respect the request headers, timeouts and all other metadata.

In practice, the situation can be more complicated depending on the client application. As discussed in the previous chapter, ShareLatex services were not meant to have a significant


nginx.conf (simplified)

```
# gets requests from users, forwards them to Koala
location /project/(.+)/(.*)$ {
   proxy_pass "koala_url/object/$1/callback/project_cb/$1/$2"
}

# gets requests from Koala, forwards them to web
location /project_cb/(.*)$ {
   proxy_pass "local_web_url/$1"
}
```

Figure 6.11: Redirecting object requests in nginx.

latency between them. That is also reflected in some request header parameters. For instance, in some cases after each request the connection between two services is closed. This means that a TCP handshake is required before any request. This has no serious impact if the services run in the same machine, but if there is a significant latency between them, the costs of a request are doubled.

In order to keep the connection alive, we have to update the headers of the request in Koala. Such a solution is sufficient in the case of ShareLatex to bring the costs down, but it has a certain impact on the generated traffic. Additionally, the duration of a connection is also a parameter which is specific to the application.

6.4.5 Why request redirection moves data too?

We mentioned earlier that there are two sides for the object migration, the migration of reference in Koala and the migration of the actual data within the application. While we explained how the object reference is migrated between Koala nodes, we did not provide specific details about the actual data migration. Here we consider how this works in ShareLatex where the object data is the project data.

Due to ShareLatex implementation and the way we split the services between core and edge, a request for a project in a certain edge, brings the project data to the instances of that edge. This happens because the project data is stored in the *docstore* service which is a centralized service.

This service stores periodically the latest version of a project. When a request for a project goes to a *web* service, this one will look first in the local *Redis* if it is stored there, otherwise it will retrieve it from the *docstore* and store it on the local *Redis* which then will provide access to other services.

This mechanism is unreliable for various reasons. During the migration, there might be data in the old location which were not saved (not flushed to the *docstore*). Additionally, data in the old location are not actually moved from there but rather copied to the new location. In the event of a migration back to the original edge the consistency of the project is not guaranteed. For instance, let us consider the following scenario: version v_1 of project P is stored at the *Redis* of *edge1* but then the project is moved to *edge2*. At this new location the project is updated to another version v_2, but after a while it is moved back to *edge1*. At this moment the *Redis* of *edge1* will still have the v_1 and it is uncertain if it will get the v_2 from the *docstore* or users will still see v_1. Such a scenario can easily lead to an inconsistent state of the project.

In short, consistency is not possible to be guaranteed just by request redirection, without an explicit mechanism within the application to properly handle data migration. However, the current default behaviour of ShareLatex allows us to run experiments on the impact of project migration on the perceived latencies, without focusing on their consistency.

6.5 Evaluation

In this section we describe the experiments we conducted using Koala as a service discovery mechanism for the edgified ShareLatex application. We mainly focus on the ability of Koala to adapt the location of a project to the location of its users and show the impact it has on their perceived latency and load distribution. Additionally, we show the overhead Koala introduces as a middleware layer.

6.5.1 The moving project

In the first experiment we show how Koala adapts to user mobility. We consider a project shared by two users. While the location of one of the users stays unchanged, the one of the other changes continuously. We monitor the perceived latency when the second user changes location and after Koala adapts the location of the project to the new location of its users.

Experiment setup. We consider a distributed environment of four edge locations and one core location. Each location corresponds to a Grid'5000 [63] server node 2 Intel Xeon E5-2630 v3 CPUs and 128GB of RAM. Koala and registrator are deployed as Docker containers in all five locations. In the *core* location we deploy all ShareLatex services, while in the edge locations only the edge-suited services as proposed in the previous chapter: *web*, *document-updater*, *real-time*, *track-changes*, *spelling*, *clsi* and *chat*. Users are simulated using the Locust [104] tool described in the previous chapter.

Based on information provided by an Internet Service Provider in the EU [146], we consider a three-layers hierarchical structure for our locations as shown in Figure 6.12. The first layer (L1) consists of the *core* location. The second layer (L2) consists of servers between the core and the end users; in our case one location, *reg1*. In this experiment we omit a potential L2 location between core and *edge3* because given the specific use case taken in consideration, it is irrelevant for the results of the experiment. Finally, the third layer (L3) consists of servers which are the closest to the end users, *edge1*, *edge2* and *edge3*. The latencies considered here are emulated using the traffic control (tc) tool and the latency ratios between these locations are also based on the model obtained in [146].

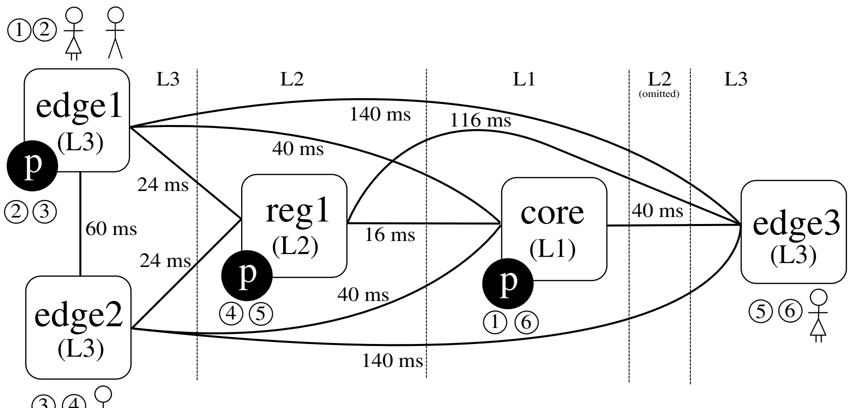
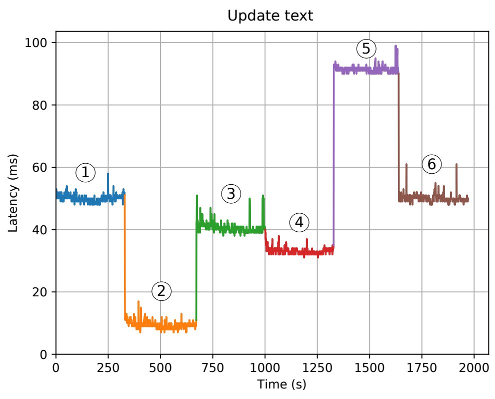
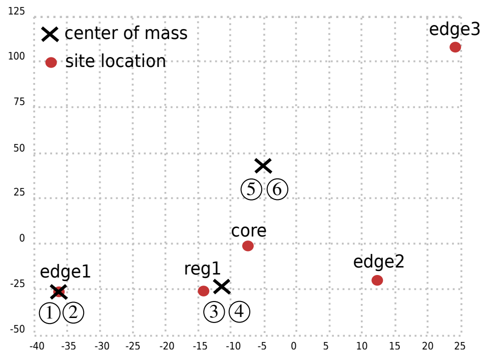


Figure 6.12: Moving project experiment setup.

Experiment description and results. We consider two users which share the same project *p*. Every second, each user issues a text update (writing) request. We monitor the end-to-end latency from the moment the text is updated from one user to the moment the update appears in the screen of the other. This latency is composed of a static application processing time (of



(a) Update text latency.



(b) Vivaldi coordinates and center of mass.

Figure 6.13: Moving project results.

≈ 10 ms), which is present regardless of the users location, and the network delay, which varies as users change their location. We plot the end-to-end latencies in Figure 6.13a, but we focus on the dynamic network part of the latency for the explanation. Additionally, we show the Vivaldi coordinates of our locations and the center of mass of users in different stages in Figure 6.13b.

Initially, these two users are both located in *edge1*, while the project *p* is located in the core (① in Figure 6.12). This means that the latency for updating the text is roughly the RTT between *edge1* and *core* (≈ 40 ms + 10 ms processing ≈ 50 ms) (① in Figure 6.13a). Given that all the requests for project *p* come from the same location, *edge1*, that location is also the center of mass (① in Figure 6.13b), and therefore Koala moves the project there ②. This is reflected also in the latency graph where we notice that the latency drops significantly, to the cost of processing the requests by the application (≈ 10 ms). At this point we decide to move one of the users to *edge2* while the project is still at *edge1* ③. We notice that the latency increases by roughly half of the RTT between *edge1* and *edge2* (≈ 60/2 ms + 10 ms ≈ 40 ms). When the number of requests reaches the threshold at which the location of the project needs to be reevaluated, Koala calculates that the center of mass between the two users is rather *reg1* and decides to move the project there ④. This reduces the latency to half of the RTT of the path *edge1*, *reg1*, *edge2* (≈ 24/2 ms + 24/2 ms + 10 ms ≈ 34 ms). After a while, we decide to move the user of *edge2* to *edge3* while the project is still in *reg1* ⑤. This results in an increase in latency to half or RTT the path *edge1*, *reg1*, *edge3* (24/2 ms + 116/2 ms + 10 ms ≈ 90 ms). When the threshold is reached again Koala will evaluate that the center of mass between users in now again the *core* and it will move the project there ⑥. This results in the initial latency ⑥ even though the users are in different edges, the latency to the core from each edge is the same. In conclusion, Koala reacts to user mobility by moving the project to a location which reduces the perceived latency for its users.

6.5.2 Project distribution

In this experiment we depict again Koala's ability to migrate the project to a location which improves user experience. However, in this case we rather focus on how the request load is distributed between different edges based on their users. We consider the number of projects in a location as an indicator of load, as the number of requests to the project instance group in a specific location is proportional to the number of projects in that location.

Experiment setup. The setup for this experiment is identical to the previous one (Figure 6.14). We consider the same 5 locations with the same latencies between nodes. Once again, users are connected only on L3 locations: *edge1*, *edge2* and *edge3*. L1 (*core*) and L2 (*reg1*)

Project	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
Users	u1, u5, u6	u2	u1, u3	u1, u4	u5, u6	u6	u4, u7	u5, u8, u9	u5, u8, u9	u8, u10
User locations	e1, e2, e2	e1	e1, e1	e1, e1	e2, e2	e2	e1, e2	e2, e3, e3	e2, e3, e3	e3, e3
Ideal site(s)	e2, **r1**, e1	e1	e1	e1	e2	e2	r1, e1, **e2**	**e3**, core, e2	e3, **core**, e2	e3

Table 6.1: Distribution of projects, users and ideal site placements.

locations can host the application but are not reached directly from the users, only through redirection.

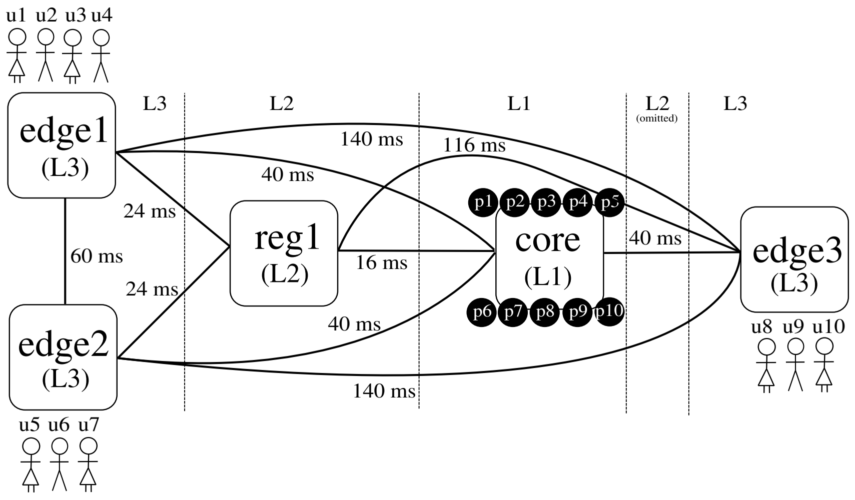


Figure 6.14: Project distribution experiment setup.

Experiment description and results. In this experiment we consider 10 users, distributed in different edges as shown in Figure 6.14. Additionally, we consider 10 projects which initially are located in the *core*. Each project is most commonly shared between two users, but in some cases it can be shared by three users or just owned by a single user as shown in Table 6.1. Moreover, projects are shared between users which are most commonly in the same location, but projects shared between remote users are present too.

In this experiment users are stationary but the projects should move in order to reflect the location of their users. A user can edit only one randomly chosen project at a time and the duration of an editing session is also random. Therefore, there is no guarantee that if two users share the same projects, they make roughly the same number of request during a specific period. The location of the project can change depending on the location of the user who edits it the most during a certain period. We can only roughly predict the location of a project by considering all the possible locations it can be at any time (last row in Table 6.1).

We simulate the 10 concurrent users in various edge locations and monitor the location of the projects at different snapshots with a duration of 1000 seconds between each other. We run this experiment until the project with a deterministic destination reach this destination (Figure 6.15).

We notice right from the first snapshot that most of the projects move away from the *core* towards their users. In general, projects whose final destination is deterministic such as *p2*-*p5* and *p10*, move right away to their expected locations, while *p6* takes a much longer time. This can be due to the fact that the only user of p6 does not choose that project until the final part of the experiment. On the other hand, we notice that projects with indeterministic locations oscillate between these locations. For instance, *p7* moves between *reg1* and *edge2*, while *p8* and *p9* between *edge3* and *core*. In any case, the locations of all the projects at the end of this experiment correspond to one of the options predicted in Table 6.1 (marked in bold).

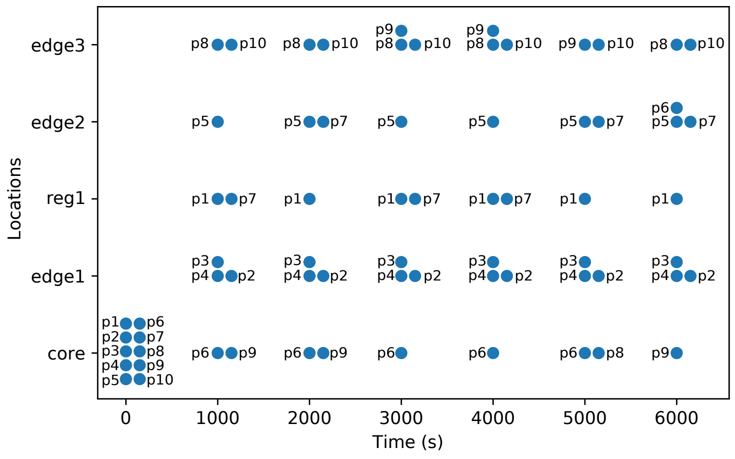


Figure 6.15: Project distribution over time.

6.5.3 Overhead

As a middleware layer between services, Koala adds some overhead to the application performance. In this section we evaluate this overhead by comparing the response time of some representative services in settings with and without Koala.

Experiment setup. Isolating the overhead, and more generally, comparing a middleware supporting edgification with another one working in centralized settings is difficult. Yet we chose two similar setups (Figure 6.16). The first one is a standard centralized deployment where all services are running in the *core* and there is a 50 ms RTT between the user and these services (left side of the figure). In the second deployment, we consider an edge (*edge1*) and the *core*. In the *core* we deploy all the services, while in *edge1* only the edge services as in the previous experiments. In both locations we deploy Koala, and the latency between them is again 50 ms.

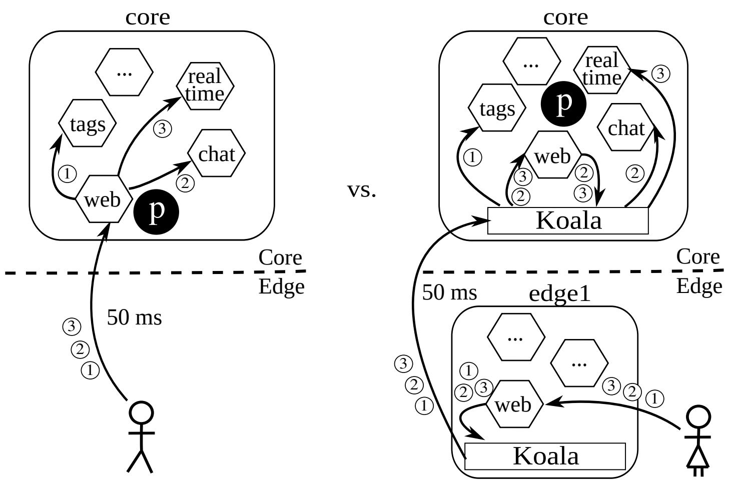


Figure 6.16: Overhead setup.

Experiment description and results. In both settings described above, the final service instance which responds to the user request is in *core*. In the centralized settings the request is first sent to the *web* service and then forwarded to the right service, while in the second setting the request goes first through the local *web* which then proxies the request to the *edge1* Koala which in turn forwards it to the *core* Koala and then to the right instance.

We distinguish three kinds of requests, two HTTP requests and one WebSocket request. From the HTTP requests we consider one which is not specific to an object (to a project), such as tags ①, and one which is related to an object such as a chat request ②. As a WebSocket request we use the update text (writing) ③, which is also an object specific request.

As explained before, redirection for services and for object requests is slightly different. While for the services there is only one interaction with Koala (follow ①), for objects there are two, once to locate the object using the reverse-proxy and one to locate the actual service (follow ② and ③). Therefore, we expect a slightly higher overhead for object-related requests.

The response times of the three requests with and without Koala are shown in Figure 6.17. When using Koala we consider both cases, the first call to the service (no cache), which requires a lookup, and the successive calls, where the results of the lookup are cached. The lookup cost can vary depending on the number of hops to destination and can be significant, but it occurs quite rarely. The case when caching is used represents better the added cost of Koala when the destination instance is already known. The WebSocket requests occur within a connection, therefore caching does not apply.

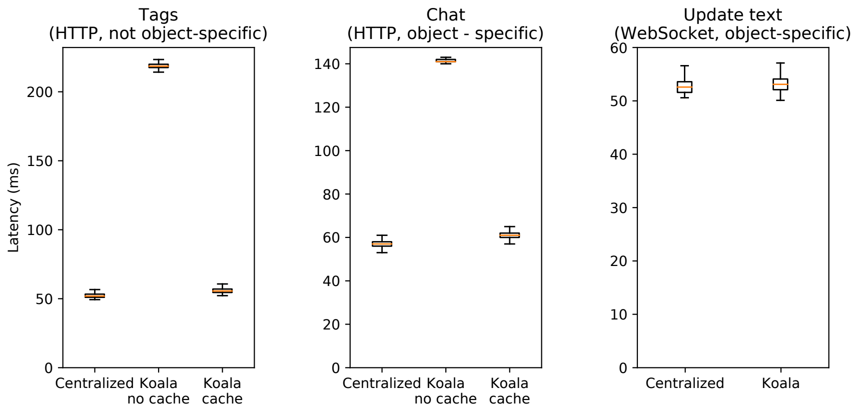


Figure 6.17: Overhead results for three requests.

From the results in Figure 6.17 we notice that the difference between the centralized setting and the one with Koala with cache are very small in all cases. In case of non object-specific requests this difference is ≈ 3 ms, while for object requests it is ≈ 4 ms, if we compare the medians. For WebSocket requests this difference is even smaller ≈ 1 ms, which can be explained by the fact that this protocol is much more lightweight than HTTP. In conclusion, if we do not consider the lookup cost, which occurs rarely, the overhead of Koala is in general insignificant.

6.6 Discussion

In this chapter we introduced the Koala service discovery prototype and considered ShareLatex as a use case application for illustrating the integration of an application with Koala. In this section we discuss some key points regarding the generalization of Koala, so that it can be adopted by any other application. More specifically, we discuss the changes needed to make Koala more generic, and what rules that edge applications should comply with for a flawless integration with Koala.

6.6.1 Improvements on Koala

Currently, Koala is quite generic in the way an application interacts with it. Even though ShareLatex requests were used to test the implementation, there is no code specific to these requests. However, when it comes to object redirection an interaction with a reverse-proxy is required and relying on the existence of this proxy is sub-optimal.

A way to improve this is to provide a Koala reverse-proxy which runs together with the service and facilitates the interaction with Koala service discovery. This tool could provide a user-friendly configuration mechanism for defining requests which create, get or delete an object, as well as describing where the object identifier is in the request, through regular expressions. Such configuration can be tested and verified through dry runs of the requests. In this way we can define a standard way this information is communicated between the two Koala modules without relying on third-party software.

6.6.2 Guidelines for edge applications

During the edgification of ShareLatex and its integration with Koala we observed a few features which are relevant to enable the edgification of an application. While in many cases ShareLatex happens to comply with these guidelines, in other occasions we needed to find a workaround which did not come without costs, in terms of reduced benefits from edgification. We list here a few of these guidelines.

Fist of all, it is crucial for the HTTP requests on objects to comply with RESTful rules. The identifier of an object should always be part of the URL in case of retrieval, update or delete operations on objects. In case of creation, this identifier should be part of the response. Moreover, the use of the right HTTP methods, especially POST for creation and DELETE for deletion of objects are highly encouraged.

Along similar lines, general best practices, such as the twelve-factor app [160], are important for edge applications. For instance, having configurable URLs as environment variables allows redirection of service traffic without modifying any config file. Additionally, making no assumptions regarding the location of the services avoids adapting request parameters, such as timeouts and header options, to a predefined topology. Such options could be at best configurable.

On the architectural side, one of the most important guidelines to enable edgification is the modular architecture. Microservices architecture is one option but even applications which adopt it often do not fully comply with its principles. In case of ShareLatex we noticed that several services are sharing the same database, making these services not self-contained and their deployment as a single entity impossible.

Another critical guideline is the object-oriented design. The idea of an object as a data unit identified by a unique identifier should be central to the data model of the application. Similarly to services, objects should be self-contained and not have any dependency with the rest of the data. This allows a seamless transfer from one service instance to another.

Finally, the application should provide a standard API which enables this object migration and handles the transition phase. Such API should allow the middleware layer (service discovery) to specify the addresses of the destination instances where the object should migrate. Ideally, the transition should be done within the application, while the middleware is notified about the progress in order to redirect the requests to the right instances.

6.7 Comparison to related work

In Section 2.4 of the State of the Art we discussed about various microservice discovery mechanisms. In this section we discuss about how they compare to our Koala prototype. We focus on the most common tools used currently in industry for microservice discovery, which include: ZooKeeper [74], Etcd [37], Eurkea [126] and Consul [71].

ZooKeeper and Etcd were not initially designed as a service discovery registry, and thus are not optimized for this specific use case. While cluster consensus is critical in certain use cases, such as distributed synchronization, it does not scale well in large distributed environments. Specifically, in an edge environment where there is a priority for local services, a full replication of data about services running in distant edges is redundant and expensive in terms of traffic.

SmartStack [7] enriches Zookeper with a proxy (Synapse) and a health checking mechanism (Nerve). The use of a local Synapse proxy is very similar to our prototype where all the requests are redirected to a local Koala container running along other services. However, each local Synapse instance is proactively updated with the list of available instances, and at each time it re-configures the back-ends of HAproxy. In Koala, we follow a rather lazy approach based on caching. The same can be said for health checking. Koala does not try to proactively remove unavailable services, but it transparently redirects the request to working alternatives and removes the stale ones in the background. SmartStack brings ZooKeeper closer to the functionalities of Koala, but it does not solve its scalability issues and it introduces further traffic for keeping Synapse nodes updated.

Eureka introduces scalable read clusters which play the role of caches closer to the clients. This is similar to caching in Koala, but the difference is that the read clusters need to be proactively synchronized with the write ones, and the write clusters are also replicated in an eventually consistent way. Moreover, the locality aspect for instances which are not registered in the local region is not taken into account.

Similar to Koala, Consul supports datacenters and the consensus is reached only within a datacenter and not between datacenters. Requests for services in other datacenters are simply forwarded to a master node in the appropriate datacenter. Another similarity with Koala is that Consul uses Vivaldi coordinates to locate the closest service starting from a certain node. Nevertheless, Consul differs from Koala in a couple of aspects. First, it introduces a gossiping and health check mechanism which comes at high traffic cost. Second, requests for the same instance are served by different agents, and therefore it can not keep track of where the clients of a certain service or object are. This means Consul is unable to handle object relocation, which is the case with all the tools mentioned here.

In conclusion, tools like Consul, Eureka and SmartStack offer additional features like strong consistency, locking, and periodic health checks, which can be useful for certain applications but come at a high network cost, especially as the system scales up. Moreover, none of the tools above provides a platform to enable service discovery based on their data, and therefore they are unable to manage sharded services. Koala uses a lazy approach in order to deal with scalability and provides object localization and migration. In this respect, Koala provides a high-level approach for data discovery in the edge which is also an internet requirement from Internet Engineering Task Force (IETF) [75].

6.8 Conclusion

In this chapter we described the transformation of Koala DHT into a service discovery mechanism suitable for an edgified microservice-based application, specifically ShareLatex. One of the particularities of this mechanism is that it does not only deal with services with replicated databases, but also with services which database is sharded between different instances. In the later case, service discovery is about finding the service instance which contains the requested data.

We revised the concept of an *object* in service discovery as data which spans through different services, but is linked to a single location. This location is the starting point for the service discovery to locate the right services instance. We continuously monitored the location of the users requesting a certain object, and were able to determine if the actual location of the object corresponds to the location of its users. In case of a discrepancy, our system was able to suggest a new location and migrate the reference of the object to this location.

Through experiments we showed the impact object migration had on user perceived latency and the load distribution between edge locations. We tested the ability of Koala to properly adapt the object location in case of user mobility, and in case users have different activeness. Finally, we showed that the overhead Koala adds to a request is negligible in nearly all cases.

Chapter 7

Conclusion and Future Work

In this dissertation we focused on three main goals: providing building blocks for fog platform management, attaining and evaluating the fog deployment of legacy applications, and delivering a service discovery mechanism as an example of a management task based on our building block.

Regarding the fog management building block, rather than dealing with specific management tasks, we aimed at identifying functionalities which are common to many of these tasks. These included communication primitives, load distribution, scalability, resource localization, interfacing, user mobility and QoS management. Our first objective was to find techniques and devise mechanisms to provide these functionalities while complying with the principles of fog, focusing especially on locality preservation and traffic reduction. We gathered these mechanisms into an overlay network, which serves as a building block for implementing higher-level management tasks. In this way, these tasks can focus only on their specific policies and not on the entire stack of generic functionalities.

In order to provide a full end-to-end examination of the management layer, we also focused on the requirements introduced by applications running on a fog infrastructure. Given that fog-native applications are rare, infrastructure-specific and often not available (closed source), our second goal was to identify architectures and techniques for adapting a non-fog-native application, such as Sharelatex, to a fog deployment without modifying its source code. In addition, we aimed at evaluating the extent to which such deployment was beneficial for this application.

Given our overlay building block, and our newly converted fog application, our third goal was to address the requirements introduced by the application by building upon our overlay. This was materialized into a service discovery mechanism which brings all the pieces together and allows us to test and validate our prototype using real application use cases.

7.1 Summary of findings

We argued that four of the main pillars of fog management were: decentralization, simplicity, locality-awareness and traffic reduction. In order to implement these fundamental requirements, we introduced an overlay network called Koala. Koala has a simple flat structure and provides locality-awareness while avoiding any unnecessary maintenance traffic. It does this by adopting a lazy approach, which consists in using application traffic for maintaining the structure of the overlay. Our experiments showed that despite its passive maintenance, Koala provides efficient locality-aware routing and is resilient to high levels of churn.

Regarding the fog deployment of existing applications, we observed that modular architectures, such as microservices, enabled this deployment without significant changes in the source code. This allowed us to replicate and place them in various locations in the fog. However, for stateful microservices the impact replica synchronization on the application varied. Based on this impact, we defined a taxonomy of stateful microservices which serves as a guideline for determining whether they should be deployed at the edge or at the core. We used this taxonomy for the fog deployment of Sharelatex. Our experiments showed that this deployment favored the most important real-time use cases, while other use cases were negatively impacted.

During the fog deployment of Sharelatex we dealt with microservices whose state was sharded between different instances. Therefore, requests for specific data needed to be redirected to the appropriate shard able to serve that request. We observed that this kind of requests contained a unique identifier pertaining to a data unit, which we called object. We treated the localization of a specific shard as part of a more generic "object-driven" service discovery mechanism. In this way, our system keeps track not only of the location of the services but also that of the objects (data units) within them. In addition, our system tracks the location of the users requesting a specific object. Therefore, it handles user mobility by suggesting the migration of objects towards the locations of the users requesting them. This was reflected in our experiments based on real-life Sharelatex use cases, where projects continuously followed their users in order to preserve request locality and reduce latency, while having a minimal redirection overhead.

7.2 Generalization

As the demand for fog applications increases, current cloud platforms are likely to be revised for accommodating the management of more distributed infrastructures. While some of the management tasks might still remain centralized, others will require decentralization in order to deal with increased traffic and latency constraints. In the latter case, the Koala overlay can provide an initial building block for the decentralized management of these tasks. By adopting it in different contexts Koala can be gradually enriched with additional features common to various tasks.

From the application standpoint, this dissertation provides a practical guide for achieving fog deployment. Considering that fog computing is an extension of the cloud model, it is likely that many of the future fog applications would come from a cloud background, *i.e.* they will need to be ported to the fog, rather than implemented from scratch. This study, not only provides detailed instructions of how to this can be achieved, but also describes the extent to which this can be beneficial without significant code modification. Moreover, it also provides relevant guidelines for developers implementing fog applications from the ground up.

In addition to the application deployment, the fog model introduces new challenges regarding locality preservation under user mobility [124]. In this study we presented an approach for handling mobility at the platform management level by supplying it with user traffic initially directed to the application. This approach separates the monitoring of requests' origins, from the actions required to handle mobility. While the platform notifies the application when the distance between users and their resources can be improved, it is up to the application to decide on a policy to do so. This model can be adopted by future platforms to consolidate locality management and develop standard interfaces to interact with their applications.

Another challenge in fog environments is also the distribution of data along various locations. As the number of geo-distributed services increases, maintenance of a full replicated state becomes expensive. Therefore, partially replicated services and sharded services are likely to become ubiquitous. This implies that additional mechanisms for locating the instance which contains the requested data are needed. In this dissertation we introduced the concept of object discovery as an extension to service discovery, which allows applications to define units of data which are traceable and localizable between instances of the same service. This model can be a first step towards a standard interface between service discovery mechanisms and fog applications which require external management of data localization.

7.3 Strength and limitations

Strength. In addition to the contributions mentioned above, the outcome of this study provides a few practical aspects which can be relevant for its replication or extension. We extended the PeerSim simulator to support various physical network topologies, which can either be generated pragmatically or uploaded in standard graph formats. Moreover, the simulator was enhanced to support custom traffic patterns, which is useful for enabling the generation of application-based traffic patterns. Finally, we provide a flexible framework for comparing new network overlays with existing ones, using the exact same setup. Therefore, the infrastructure for future work is already in place.

Another strong point of this study is the use of an existing well-known application for evaluation and obtaining requirements. Oftentimes middleware studies use simple proof-of-concept applications which source code is fully under control and can be modified to fit the framework provided by the middleware. In this study, we did the opposite. We kept the application intact while trying to adapt our middleware to it. In this way we made sure to address relevant existing challenges.

A final important strong point is the implementation of a fully integrated prototype, which can be easily redeployed, verified and extended. As remarked in [124] many fog-related studies are only simulation-based or are not easily replicated. Our prototype instead is designed to be easily deployed in various distributed environments, without the need of specific tools. Its modular structure supports the deployment of custom fog applications, other than the provided Sharelatex.

Limitations. Given the wide scope of this study, certain aspects were difficult to address considering the limited time and the novelty of the field. While our simulation setup is flexible to accept custom configurations, our network topology and traffic pattern were generally random. As fog infrastructures maturate, their network topology models and fog application traffic patterns might become more accessible. In that case, we can integrate them to our simulator for a better evaluation of our overlay.

While simulations are limited for not being too specific, the limitations of use-case applications are that they are not too generic. In our case we focus on microservices applications which use non-encrypted messages based on REST interfaces. While most of the concepts discussed in this study can apply to alternative setups, several issues regarding security (encryption) or other messaging protocols (gRPC) are not dealt with. We consider that such features can be added gradually as more specific applications use our prototype for fog deployment.

Regarding the QoS management in the service discovery, we currently provide the user with the service instance with the lowest latency. However, this approach can be extended to a more generic one, where not only latency but also other criteria, such as bandwidth, are used for the selection. In addition, the user can provide at each request also an SLO-like specification which can help the discovery mechanism select the most appropriate instance.

Finally, given that our focus for the prototype was on integration and functionality, we did not scale the number of locations and users to those of a common web application. However, we provide all the necessary tools to do so. A real-size deployment with real users would be very useful for deriving traffic patterns which can later be injected to our simulator.

7.4 Future directions

While dealing with the aforementioned limitations can be the basis of a short term future work, in the long term we envision an expansion and integration of these ideas in a joint endeavor to provide standard fog platforms. Given the current efforts for decentralizing cloud platforms, such as OpenStack [132], our goal is to use our overlay as a flexible and adaptable alternative to current general purpose messaging systems. Adapting the Koala overlay to support the decentralization fundamental OpenStack services such as Nova compute or Glance is one of the future objectives.

Regarding the service/object discovery mechanism, a future direction is to integrate it into an existing container management platform, such as Docker swarm [46] or Kubernetes [55]. By allowing our mechanism to create and manipulate containers, it can provide full support to service replication and shard migration. Currently, our system redirects requests only to the already deployed containers. If integrated in a management platform, our system can track users and create replicas or shards in locations which would create opportunities for further reducing latencies and traffic generation. This opens a new research direction regarding policies about when containers need to be replicated, split or merged, moved or deleted.

Along the same lines, another direction is to provide an automatic deployment for a fog application. Ideally, the fog application can initially be deployed in a centralized location and gradually become distributed based on user requests. We currently provide such functionality to some extent, but our split between services which remain centralized and those which can relocate was done manually. We would like to consider ways to automatically identify the category of the service, potentially without the provider specification. Therefore, a new research line can focus on dry-run profiling by replicating user requests and analyzing the effect replication has on the application and deployment.

Finally, based on our deployment approach of externalizing data localization, a new research direction can focus on building application models for supporting this framework. As mentioned in Chapter 6, in order to make our deployment fully functional, some collaboration from the application is necessary. This research can focus on providing standard and secure interfaces for allowing the definition of objects as data units, as well as other elements needed to enable a flawless control exchange between the application and the service discovery. Standardized reverse proxies and frameworks for masking the background interactions from the users are a few examples of such collaboration.

Despite the recent advancements, Fog still remains a novel paradigm that needs to maturate over the next decade. Maturation often consists in a cyclic process of advances in the demand for fog applications and the supply of fog platforms. As the demand for fog applications increases, new platforms meeting basic requirements will emerge. This will in turn trigger the development of more applications, which will further contribute in the consolidation of platforms. In this dissertation we followed a similar cyclic approach starting with an initial platform building block, then deployed an application and learned from this deployment so that we could further enrich platform management. By providing a flexible and agile model we presented an example of an iteration of this cycle, which can serve as the basis for future iterations until a standard fog platform emerges.

Bibliography

[1] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. “Route maintenance overheads in DHT overlays”. In: *The 6th Workshop on Distributed Data and Structures (WDAS 2004)*. CONF. 2004.

[2] Karl Aberer et al. “Gridvine: Building internet-scale semantic overlay networks”. In: *International semantic web conference*. Springer. 2004, pp. 107–121.

[3] Karl Aberer et al. “P-Grid: a self-organizing structured P2P system”. In: *Sigmod Record* 32.ARTICLE (2003), pp. 29–33.

[4] Carlos M. Aderaldo et al. “Benchmark Requirements for Microservices Architecture Research”. In: *1st Intl. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. ECASE. 2017.

[5] Swati Agarwal, Shashank Yadav, and Arun Kumar Yadav. “An efficient architecture and algorithm for resource provisioning in fog computing”. In: *International Journal of Information Engineering and Electronic Business* 8.1 (2016), p. 48.

[6] Central Intelligence Agency. *The world factbook 2016*. Government Printing Office, 2016.

[7] Airbnb. *SmartStack:Service Discovery in the Cloud*. URL: `https://medium.com/airbnb-engineering/smartstack-service-discovery-in-the-cloud-4b8a080de619`.

[8] Mouna Allani, Benoît Garbinato, and Peter Pietzuch. “Chams: Churn-aware overlay construction for media streaming”. In: *Peer-to-Peer Networking and Applications* 5.4 (2012), pp. 412–427.

[9] Amazon. *Amazon Elastic Compute Cloud*. URL: `https://aws.amazon.com/ec2/`.

[10] Amazon. *Amazon S3*. URL: `https://aws.amazon.com/s3/`.

[11] Amazon. *AWS Elastic Beanstalk*. URL: `https://aws.amazon.com/elasticbeanstalk/`.

[12] Amazon. *Global Infrastructure*. URL: `https://aws.amazon.com/about-aws/global-infrastructure/`.

[13] Anders Andrae and Tomas Edler. “On global electricity usage of communication technology: trends to 2030”. In: *Challenges* 6.1 (2015), pp. 117–157.

[14] Michael Armbrust et al. “A View of Cloud Computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: `10.1145/1721654.1721672`. URL: `http://doi.acm.org/10.1145/1721654.1721672`.

[15] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. “Design and implementation of a P2P Cloud system”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 412–417.

[16] Ozalp Babaoglu et al. “Design patterns from biology for distributed computing”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 1.1 (2006), pp. 26–66.

[17] Marin Bertier et al. "Cloud Computing: Challenges, Limitations and R&D Solutions". In: 2014. Chap. Beyond the Clouds: How Should Next Generation Utility Computing Infrastructures Be Designed? ISBN: 978-3-319-10530-7. DOI: `10.1007/978-3-319-10530-7_14`.

[18] Betsy Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems.* " O'Reilly Media, Inc.", 2016.

[19] Luiz Fernando Bittencourt et al. "Towards virtual machine migration in fog computing". In: *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC).* IEEE. 2015, pp. 1–8.

[20] Flavio Bonomi et al. "Fog computing: A platform for internet of things and analytics". In: *Big data and internet of things: A roadmap for smart environments.* Springer, 2014, pp. 169–186.

[21] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing.* MCC '12. Helsinki, Finland: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: `10.1145/2342509.2342513`. URL: `http://doi.acm.org/10.1145/2342509.2342513`.

[22] Lars Braubach, Kai Jander, and Alexander Pokahr. "A novel distributed registry approach for efficient and resilient service discovery in megascale distributed systems?" In: *Comput. Sci. Inf. Syst.* 15.3 (2018), pp. 751–774. URL: `http://doiserbia.nb.rs/Article.aspx?id=1820-02141800030B`.

[23] Eric A Brewer. "Towards robust distributed systems". In: *PODC.* Vol. 7. 2000.

[24] Amos Brocco, Apostolos Malatras, and Béat Hirsbrunner. "Enabling efficient information discovery in a self-structured grid". In: *Future Generation Computer Systems* 26.6 (2010), pp. 838–846.

[25] M. Báguena et al. "Towards enabling hyper-responsive mobile apps through network edge assistance". In: *13th IEEE Annual Consumer Comm. Net. Conf.* CCNC. 2016.

[26] Matt Calder et al. "Analyzing the Performance of an Anycast CDN". In: *Proceedings of the 2015 Internet Measurement Conference.* ACM. 2015, pp. 531–537.

[27] Shai Carmi et al. "A model of Internet topology using k-shell decomposition". In: *Proceedings of the National Academy of Sciences* 104.27 (2007), pp. 11150–11154.

[28] Miguel Castro, Manuel Costa, and Antony Rowstron. "Debunking some myths about structured and unstructured overlays". In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2.* USENIX Association. 2005, pp. 85–98.

[29] Miguel Castro et al. "Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks". In: *Future Directions in Distributed Computing, Research and Position Papers.* Springer, 2003. DOI: `10.1007/3-540-37795-6_21`.

[30] Abhishek Chandra, Jon Weissman, and Benjamin Heintz. "Decentralized edge clouds". In: *IEEE Internet Computing* 17.5 (2013), pp. 70–73.

[31] M. Chiang et al. "Clarifying Fog Computing and Networking: 10 Questions and Answers". In: *IEEE Communications Magazine* 55.4 (2017), pp. 18–20. ISSN: 0163-6804. DOI: `10.1109/MCOM.2017.7901470`.

[32] Junguk Cho et al. "ACACIA: Context-aware Edge Computing for Continuous Interactive Applications over Mobile Networks". In: *12th Intl. on Conference on Emerging Networking EXperiments and Technologies.* CoNEXT. 2016.

[33] Kenneth Church, Albert G Greenberg, and James R Hamilton. "On Delivering Embarrassingly Distributed Cloud Services." In: *HotNets.* 2008, pp. 55–60.

[34] S. Clinch et al. "How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users". In: *IEEE Intl. Conference on Pervasive Computing and Communications*. PerCom. 2012.

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

[36] OpenFog Consortium et al. "OpenFog reference architecture for fog computing". In: *Architecture Working Group* (2017). URL: `https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf`.

[37] CoreOS. *Etcd - A distributed, reliable key-value store for the most critical data of a distributed system.* URL: `https://coreos.com/etcd/`.

[38] Microsoft Corporation. *Microsoft Azure*. URL: `https://azure.microsoft.com`.

[39] Kevin P Coyne and Renée Dye. "The competitive dynamics of network-based businesses". In: *Harvard business review* 76.1 (1998), pp. 99–110.

[40] Philippe Cudré-Mauroux, Karl Aberer, and Manfred Hauswirth. "The Chatty Web Approach for Global Semantic Agreements." In: *MMGPS*. 2003.

[41] Frank Dabek et al. "Vivaldi: A Decentralized Network Coordinate System". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '04. Portland, Oregon, USA: ACM, 2004, pp. 15–26. ISBN: 1-58113-862-8. DOI: `10.1145/1015467.1015471`. URL: `http://doi.acm.org/10.1145/1015467.1015471`.

[42] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[43] Sanjay K Dhurandher et al. "Using bee algorithm for peer-to-peer file searching in mobile ad hoc networks". In: *Journal of Network and Computer Applications* 34.5 (2011), pp. 1498–1508.

[44] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: `10.1007/BF01386390`. URL: `http://dx.doi.org/10.1007/BF01386390`.

[45] *djb2*. URL: `http://www.cse.yorku.ca/~oz/hash.html`.

[46] *Docker - Build, Ship, and Run Any App, Anywhere.* URL: `https://www.docker.com/`.

[47] Marco Dorigo, Vittorio Maniezzo, Alberto Colorni, et al. "Ant system: optimization by a colony of cooperating agents". In: *IEEE Transactions on Systems, man, and cybernetics, Part B: Cybernetics* 26.1 (1996), pp. 29–41.

[48] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[49] Dropbox. *Cloud File Sharing and Storage for your Business.* URL: `https://www.dropbox.com/`.

[50] Mohammed S Elbamby et al. "Toward low-latency and ultra-reliable virtual reality". In: *IEEE Network* 32.2 (2018), pp. 78–84.

[51] *Ethernity*. URL: `https://ethernity.cloud/whitepaper/ETHERNITY_whitepaper.pdf`.

[52] D. Fesehaye et al. "Impact of Cloudlets on Interactive Mobile Cloud Applications". In: *IEEE 16th Intl. Enterprise Distributed Object Computing Conference*. EDOC. 2012.

[53] Agostino Forestiero et al. "Self-chord: a bio-inspired P2P framework for self-organizing distributed systems". In: *IEEE/ACM Transactions on Networking (TON)* 18.5 (2010), pp. 1651–1664.

[54] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure.* Elsevier, 2003.

[55] Cloud Native Computing Foundation. *Kubernetes.* URL: `https://kubernetes.io/`.

[56] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. "Third International COST264 Workshop". In: 2001. Chap. Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. ISBN: 978-3-540-45546-2. DOI: `10.1007/3-540-45546-9_4`.

[57] Pedro Garcia Lopez et al. "Edge-centric computing: Vision and challenges". In: *ACM SIGCOMM Computer Communication Review* 45.5 (2015), pp. 37–42.

[58] Pedro Garcia Lopez et al. "Edge-centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015).

[59] Martin Garriga. "Towards a taxonomy of microservices architectures". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2017, pp. 203–218.

[60] Gartner. *Forecast: Public Cloud Services, Worldwide, 2016-2022, 4Q18.* URL: `https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g`.

[61] Raffaele Giordanelli, Carlo Mastroianni, and Michela Meo. "Bio-inspired P2P systems: The case of multidimensional overlay". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7.4 (2012), p. 35.

[62] Sarunas Girdzijauskas et al. "Fuzzynet: Ringless routing in a ring-like structured overlay". In: *Peer-to-Peer Networking and Applications* 4.3 (2011), pp. 259–273. ISSN: 1936-6450. DOI: `10.1007/s12083-010-0081-3`. URL: `https://doi.org/10.1007/s12083-010-0081-3`.

[63] *Grid5000.* URL: `https://www.grid5000.fr/`.

[64] *gRPC - A High-Performance, Open-Source Universal RPC Framework.* URL: `http://www.grpc.io/`.

[65] Indranil Gupta et al. "Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead". In: *Second International Workshop on Peer-to-Peer Systems (IPTPS'03)*. Springer, 2003.

[66] Raluca Halalai et al. "ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service". In: *33rd IEEE Intl. Symposium on Reliable Distributed Systems*. SRDS. 2014.

[67] Michael Han. *ZooKeeper at Twitter.* URL: `https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/zookeeper-at-twitter.html`.

[68] *HAProxy -The Reliable, High Performance TCP/HTTP Load Balancer.* URL: `http://www.haproxy.org/`.

[69] Nicholas J. A. Harvey et al. "SkipNet: A Scalable Overlay Network with Practical Locality Properties". In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS'03. Seattle, WA: USENIX Association, 2003, pp. 9–9. URL: `http://dl.acm.org/citation.cfm?id=1251460.1251469`.

[70] R. Hasan et al. "A survey of peer-to-peer storage techniques for distributed file systems". In: *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*. Vol. 2. 2005, 205–213 Vol. 2. DOI: `10.1109/ITCC.2005.42`.

[71] HashiCorp. *Consul.* URL: `https://www.consul.io/`.

[72] Inc. Heroku. *Heroku Platform.* URL: `https://www.heroku.com/`.

[73] Xueshi Hou et al. “Vehicular fog computing: A viewpoint of vehicles as the infrastructures”. In: *IEEE Transactions on Vehicular Technology* 65.6 (2016), pp. 3860–3873.

[74] Patrick Hunt et al. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: `http://dl.acm.org/citation.cfm?id=1855840.1855851`.

[75] Internet Engineering Task Force (IETF). *Overview of Edge Data Discovery*. URL: `https://tools.ietf.org/id/draft-mcbride-edge-data-discovery-overview-00.html`.

[76] *iExec*. URL: `https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf`.

[77] Cisco Global Cloud Index and CV Cisco Visual Networking Index. “Forecast and Methodology, 2016–2021; White Paper; Cisco Systems”. In: *Inc.: San Jose, CA, USA* (2017).

[78] *Industrial Internet Consortium*. URL: `https://www.iiconsortium.org/`.

[79] GSMA Intelligence. “Global mobile trends 2017”. In: *GSMA, September* (2017).

[80] Márk Jelasity and Ozalp Babaoglu. “Third International Workshop on Engineering Self-Organising Systems (ESOA 2005)”. In: ed. by Sven A. Brueckner et al. Springer, 2006. Chap. T-Man: Gossip-Based Overlay Topology Management, pp. 1–15. ISBN: 978-3-540-33352-4. DOI: `10.1007/11734697_1`.

[81] Márk Jelasity et al. “Gossip-based peer sampling”. In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), p. 8.

[82] Yuh-Jzer Joung and Jiaw-Chang Wang. “Chord2: A two-layer Chord for reducing maintenance overhead via heterogeneity”. In: *Computer Networks* 51.3 (2007), pp. 712–731.

[83] Andreas Kapsalis et al. “A cooperative fog approach for effective workload balancing”. In: *IEEE Cloud Computing* 4.2 (2017), pp. 36–45.

[84] Peter Kersch et al. “Stochastic maintenance of overlays in structured P2P systems”. In: *Computer Communications* 31.3 (2008), pp. 603–619.

[85] Jon Kleinberg. “The Small-world Phenomenon: An Algorithmic Perspective”. In: *Proceedings of ACM STOC’00*. Portland, Oregon, USA: ACM, 2000, pp. 163–170. ISBN: 1-58113-184-4. DOI: `10.1145/335305.335325`.

[86] Fabius Klemm et al. “On routing in distributed hash tables”. In: *Seventh IEEE International Conference on Peer-to-Peer Computing (P2P 2007)*. IEEE. 2007, pp. 113–122.

[87] Georgia Koloniari et al. “Query Workload-aware Overlay Construction Using Histograms”. In: *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. CIKM ’05. Bremen, Germany: ACM, 2005, pp. 640–647. ISBN: 1-59593-140-6. DOI: `10.1145/1099554.1099717`.

[88] Yoram Kulbak, Danny Bickson, et al. “The eMule protocol specification”. In: *eMule project, http://sourceforge. net* (2005).

[89] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[90] Glider Labs. *Registrator*. URL: `http://gliderlabs.github.io/registrator/latest/`.

[91] Avinash Lakshman and Prashant Malik. “Cassandra: structured storage system on a p2p network”. In: *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM. 2009, pp. 5–5.

[92] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: `10.1145/359545.359563`.

[93] Adrien Lebre, Jonathan Pastor, Discovery Consortium, et al. "The DISCOVERY Initiative-Overcoming Major Limitations of Traditional Server-Centric Clouds by Operating Massively Distributed IaaS Facilities". In: (2015).

[94] Sergey Legtchenko et al. "RelaxDHT: A Churn-Resilient Replication Strategy for Peer-to-Peer Distributed Hash Tables". In: *ACM Transactions on Autonomous and Adaptive Systems* 7.2 (2012), 28:1–28:18. DOI: 10.1145/2240166.2240178.

[95] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *International Conference on Dependable Systems and Networks (DSN'07).* 2007. DOI: 10.1109/DSN.2007.56.

[96] Kfir Lev-Ari et al. "Modular Composition of Coordination Services". In: *2016 Usenix Annual Technical Conference.* ATC. 2016.

[97] Chao Li et al. "Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management". In: *ACM Comput. Surv.* 51.2 (Apr. 2018).

[98] Tonglin Li et al. "ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table". In: *27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2013).* Cambridge, USA, May 2013, pp. 775–787. DOI: 10.1109/IPDPS.2013.110.

[99] Albert van der Linde et al. "Legion: Enriching Internet Services with Peer-to-Peer Interactions". In: *26th Intl. Conf. on World Wide Web.* WWW. 2017.

[100] Google LLC. *Google APP Engine.* URL: https://cloud.google.com/compute/.

[101] Google LLC. *Google Compute Engine.* URL: https://cloud.google.com/compute/.

[102] Google LLC. *Google Suite.* URL: https://gsuite.google.com/.

[103] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. "eQuus: A provably robust and locality-aware peer-to-peer system". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06).* IEEE. 2006, pp. 3–11.

[104] *Locust.* URL: https://www.locust.io.

[105] LogicMonitor. *Cloud Vision 2020: The Future of the Cloud Study.* URL: https://www.logicmonitor.com.

[106] Apostolos Malatras. "State-of-the-art survey on P2P overlay networks in pervasive computing environments". In: *Journal of Network and Computer Applications* 55 (2015), pp. 1 –23. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.04.014. URL: http://www.sciencedirect.com/science/article/pii/S1084804515000879.

[107] Dahlia Malkhi, Moni Naor, and David Ratajczak. "Viceroy: A scalable and dynamic emulation of the butterfly". In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing.* ACM. 2002, pp. 183–192.

[108] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. "Symphony: Distributed Hashing in a Small World". In: *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems.* Seattle, USA, Mar. 2003.

[109] Y. Mao et al. "A Survey on Mobile Edge Computing: The Communication Perspective". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2322–2358. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2745201.

[110] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. "DHT routing using social links". In: *International Workshop on Peer-to-Peer Systems.* Springer. 2004, pp. 100–111.

[111] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011.

[112] Laurent Massoulié, A-M Kermarrec, and Ayalvadi J Ganesh. "Network awareness and failure resilience in self-organizing overlay networks". In: *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* IEEE. 2003, pp. 47–55.

[113] Miguel Matos et al. "CLON: Overlay Networks and Gossip Protocols for Cloud Environments". In: *On the Move to Meaningful Internet Systems: OTM 2009: Proceedings of Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009.* Ed. by Robert Meersman, Tharam Dillon, and Pilar Herrero. Springer, 2009. ISBN: 978-3-642-05148-7. DOI: `10.1007/978-3-642-05148-7_41`.

[114] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *First International Workshop on Peer-to-Peer Systems (IPTPS'02).* Springer, 2002, pp. 53–65.

[115] Christopher Meiklejohn and Peter Van Roy. "Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs". In: *1st Workshop on Principles and Practice of Consistency for Distributed Data.* PaPoC. 2015.

[116] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[117] Alan Mislove and Peter Druschel. "Providing administrative control and autonomy in structured peer-to-peer overlays". In: *International Workshop on Peer-to-Peer Systems.* Springer. 2004, pp. 162–172.

[118] *MongoDB.* URL: `https://www.mongodb.com/`.

[119] Luiz Rodolpho Monnerat and Claudio L Amorim. "D1HT: a distributed one hop hash table". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium.* IEEE. 2006, 10–pp.

[120] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. "Service-Oriented Programming with Jolie". In: *Web Services Foundations.* Ed. by Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel. Springer, 2014, pp. 81–107.

[121] Fabrizio Montesi and Janine Weber. "Circuit breakers, discovery, and API gateways in microservices". In: *arXiv preprint arXiv:1609.05830* (2016).

[122] Alberto Montresor and Márk Jelasity. "PeerSim: A Scalable P2P Simulator". In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09).* Sept. 2009.

[123] Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. "Chord on demand". In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05).* IEEE. 2005, pp. 87–94.

[124] Carla Mouradian et al. "A Comprehensive Survey on Fog Computing: State-of-the-art and Research Challenges". In: *CoRR* (2017).

[125] *Neo4J Graph Platform.* URL: `https://neo4j.com/`.

[126] Netflix. *Eureka 2.0.* URL: `https://github.com/Netflix/eureka/wiki/Eureka-2.0-Architecture-Overview`.

[127] Netflix. *Ribbon.* URL: `https://github.com/Netflix/ribbon`.

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

[129] TS Eugene Ng and Hui Zhang. "Predicting Internet network distance with coordinates-based approaches". In: *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 1. IEEE. 2002, pp. 170–179.

[130] Shadi A. Noghabi et al. "Steel: Simplified Development and Deployment of Edge-Cloud Applications". In: *10th USENIX Workshop on Hot Topics in Cloud Computing.* HotCloud. 2018.

[131] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[132] *Open source software for creating private and public clouds.* URL: `https://www.openstack.org/`.

[133] A. Papageorgiou, B. Cheng, and E. Kovacs. “Real-time data reduction at the network edge of Internet-of-Things systems”. In: *11th International Conference on Network and Service Management (CNSM)*. 2015.

[134] Mike P Papazoglou and Willem-Jan Van Den Heuvel. “Service oriented architectures: approaches, technologies and research issues”. In: *The VLDB journal* 16.3 (2007), pp. 389–415.

[135] *pcap.* URL: `https://www.npmjs.com/package/pcap`.

[136] C Greg Plaxton, Rajmohan Rajaraman, and Andrea W Richa. “Accessing nearby copies of replicated objects in a distributed environment”. In: *Theory of computing systems* 32.3 (1999), pp. 241–280.

[137] Fatemeh Rahimian et al. “Vitis: A gossip-based hybrid overlay for internet-scale publish/-subscribe enabling rendezvous routing in unstructured overlay networks”. In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 746–757.

[138] Venugopalan Ramasubramanian and Emin Gün Sirer. “Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays”. In: *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*. San Francisco, USA, Mar. 2004, pp. 99–112.

[139] Michael Rambold et al. “Towards autonomic service discovery a survey and comparison”. In: *2009 IEEE International Conference on Services Computing*. IEEE. 2009, pp. 192–201.

[140] Sylvia Ratnasamy et al. *A scalable content-addressable network.* Vol. 31. 4. ACM, 2001.

[141] *Redis - REmote DIctionary Server.* URL: `https://redis.io/`.

[142] Ariyattu C. Resmi and François Taiani. “Fluidify: Decentralized Overlay Deployment in a Multi-cloud World”. In: *Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. Ed. by Alysson Bessani and Sara Bouchenak. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-19129-4. DOI: `10.1007/978-3-319-19129-4_1`.

[143] Sean Rhea et al. “Handling churn in a DHT”. In: Report No. UCB/CSD-03-1299, University of California, also Proc. USENIX ... 2003.

[144] Matei Ripeanu. “Peer-to-peer architecture case study: Gnutella network”. In: *Proceedings first international conference on peer-to-peer computing*. IEEE. 2001, pp. 99–100.

[145] Antony I. T. Rowstron and Peter Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Proceedings of IFIP/ACM Middleware’01*. 2001, pp. 329–350. ISBN: 3-540-42800-3.

[146] Sanhaji A. (Orange Labs Networks, France). Private communication. 2019.

[147] Mahadev Satyanarayanan et al. “The case for vm-based cloudlets in mobile computing”. In: *IEEE pervasive Computing* 4 (2009), pp. 14–23.

[148] Sabina Serbu, Pascal Felber, and Peter Kropf. "HyPeer: Structured Overlay with Flexible-Choice Routing". In: *Computer Networks* 1 (2011). DOI: `10.1016/j.comnet.2010.09.006`.

[149] SkyNet services. *SkyDNS: DNS service discovery for etcd.* URL: `https://github.com/skynetservices/skydns`.

[150] *Sharelatex - A Web-based Collaborative LaTeX Editor.* URL: `https://www.sharelatex.com`.

[151] Pradip Kumar Sharma, Mu-Yen Chen, and Jong Hyuk Park. "A software defined fog node based distributed blockchain cloud architecture for IoT". In: *IEEE Access* 6 (2017), pp. 115–124.

[152] W. Shi et al. "Edge Computing: Vision and Challenges". In: vol. 3. 5. 2016.

[153] Georgos Siganos, Sudhir Leslie Tauro, and Michalis Faloutsos. "Jellyfish: A conceptual model for the as internet topology". In: *Journal of Communications and Networks* 8.3 (2006), pp. 339–350.

[154] Vladimir Stantchev et al. "Smart items, fog and cloud computing as enablers of servitization in healthcare". In: *Sensors & Transducers* 185.2 (2015), p. 121.

[155] Ion Stoica et al. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of ACM SIGCOMM'01.* 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: `10.1145/383059.383071`.

[156] Roshan Sumbaly et al. "Serving large-scale batch computed data with project voldemort". In: *Proceedings of the 10th USENIX conference on File and Storage Technologies.* USENIX Association. 2012, pp. 18–18.

[157] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* Prentice-Hall, 2007.

[158] Bo Tang et al. "A hierarchical distributed fog computing architecture for big data analysis in smart cities". In: *Proceedings of the ASE BigData & SocialInformatics 2015.* ACM. 2015, p. 28.

[159] Jörg Thalheim et al. "Sieve: Actionable Insights from Monitored Metrics in Distributed Systems". In: *18th ACM/IFIP/USENIX Middleware Conference.* 2017.

[160] *The Twelve-Factor App.* URL: `https://12factor.net/`.

[161] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. "A Survey of DHT Security Techniques". In: *ACM Comput. Surv.* 43.2 (Feb. 2011), 8:1–8:49. ISSN: 0360-0300. DOI: `10.1145/1883612.1883615`. URL: `http://doi.acm.org/10.1145/1883612.1883615`.

[162] Karima Velasquez et al. "Fog orchestration for the Internet of Everything: state-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 9.1 (2018), p. 14. ISSN: 1869-0238. DOI: `10.1186/s13174-018-0086-3`. URL: `https://doi.org/10.1186/s13174-018-0086-3`.

[163] M. Villari et al. "Osmotic Computing: A New Paradigm for Edge/Cloud Integration". In: *IEEE Cloud Computing* (2016).

[164] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and systems Management* 13.2 (2005), pp. 197–217.

[165] S. Wang et al. "A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications". In: *IEEE Access* 5 (2017).

[166] B. M. Waxman. "Routing of multipoint connections". In: *IEEE Journal on Selected Areas in Communications* 6.9 (1988).

[167] Barry Wellman and Caroline Haythornthwaite. *The Internet in everyday life*. John Wiley & Sons, 2008.

[168] W. Wu et al. "LDHT: Locality-aware Distributed Hash Tables". In: *2008 International Conference on Information Networking*. 2008, pp. 1–5. DOI: `10.1109/ICOIN.2008.4472811`.

[169] Zhichen Xu, Mallik Mahalingam, and Magnus Karlsson. "Turning Heterogeneity into an Advantage in Overlay Routing". In: *Proceedings of IEEE INFOCOM 2003*. San Francisco, USA.

[170] Shanhe Yi, Cheng Li, and Qun Li. "A Survey of Fog Computing: Concepts, Applications and Issues". In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata '15. Hangzhou, China: ACM, 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: `10.1145/2757384.2757397`. URL: `http://doi.acm.org/10.1145/2757384.2757397`.

[171] Jesse Yli-Huumo et al. "Where is current research on blockchain technology?—a systematic review". In: *PloS one* 11.10 (2016), e0163477.

[172] Ashkan Yousefpour et al. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Journal of Systems Architecture* (2019).

[173] Marat Zhanikeev. "A cloud visitation platform to facilitate cloud federation and fog computing". In: *Computer* 48.5 (2015), pp. 80–83.

[174] Ben Y Zhao et al. "Brocade: Landmark routing on overlay networks". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 34–44.

[175] Ben Y Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

[176] Hao Zhuang, Rameez Rahman, and Karl Aberer. "Decentralizing the cloud: How can small data centers cooperate?" In: *14-th IEEE International Conference on Peer-to-Peer Computing*. Ieee. 2014, pp. 1–10.

Résumé en français

Le Cloud

Héberger sa propre infrastructure de calcul est devenu compliqué en raison de coûts énergétiques et de complexité de gestion accrus. Les nouvelles entreprises sont de même confrontées à de nombreux obstacles au moment de leur création, et ont besoin d'un investissement initial important pour être compétitives. La possession d'infrastructures de calcul a rapidement été remplacée par le transfert des calculs vers des infrastructures de type Cloud. Le cloud computing est devenu l'environnement de déploiement privilégié de la majorité des applications distribuées nécessitant une puissance de calcul importante. Fondé sur les principes de l'informatique utilitaire, le Cloud fournit un modèle pour la mise à disposition d'infrastructures et de services informatiques à la demande. Les caractéristiques de base de l'informatique dans le Cloud sont les suivantes : service à la demande, accès via le réseau, mise en commun des ressources, élasticité et mesurabilité. L'aspect "à la demande" implique que les utilisateurs peuvent obtenir les ressources nécessaires sans les demander systématiquement au fournisseur. Les ressources obtenues sont accessibles via des protocoles réseau bien connus pris en charge par la plupart des périphériques. Ces ressources sont partagées entre les utilisateurs : lorsqu'une ressource est libérée par un utilisateur, celle-ci devient immédiatement disponible pour les autres utilisateurs. L'élasticité implique que, pour une application, la quantité de ressource qui lui est dédiée puisse être automatiquement augmentée ou réduite en fonction de la charge de travail. Enfin, le cloud utilise un système de mesure de l'utilisation des ressources, permettant le paiement à l'utilisation.

Les chiffres et les limites

Le cloud est l'un des marchés dont la croissance est la plus rapide. Il devrait maintenir une croissance régulière supérieure à 14% jusqu'en 2022. Selon un rapport de Cisco, d'ici 2021, 94% de l'informatique au niveau global sera traitée dans des centres de données dans le cloud. Le trafic mondial annuel dans le cloud de 2016 à 2021 augmentera de 3,3 fois et les données stockées dans le nuage seront multipliées par 4.
Bien que la croissance continue du cloud semble loin d'atteindre un plateau, il convient d'examiner ses effets sur l'infrastructure actuelle. L'infrastructure de bout en bout comprend trois éléments principaux : les centres de données, le réseau (liaisons, routeurs, commutateurs) et les machines ou périphériques de l'utilisateur. Étant donné que les centres de données appartiennent à des sociétés de cloud bénéficiant directement de cette croissance, leur réaction sera probablement plus appropriée. En effet, le rapport de Cisco prévoit que les centres de données de très grande taille domineront le paysage du cloud public d'ici 2021. Cependant, les énormes centres de données nécessitent une consommation d'énergie extrêmement élevée. En 2030, la consommation des centres de données représentera jusqu'à 13% de la consommation mondiale, contre 1% en 2010. À moins de se tourner de façon drastique vers les énergies vertes, la croissance des centres de données ne semble pas durable.
En ce qui concerne les réseaux, la réponse à la croissance du cloud est moins immédiate. Différents segments de réseau sont détenus et gérés par différents acteurs, ce qui crée plusieurs problèmes. Une réglementation inadéquate et une dynamique concurrentielle entre les entreprises de réseaux peuvent entraîner des infrastructures ou des configurations non optimales. De plus, les investissements dans les réseau nécessitent une intervention capillaire dans l'infrastructure, ce qui est coûteux et comporte de

nombreuses barrières techniques (creusement de tranchées, autorisations gouvernementales). Cela signifie que les équipements réseaux peuvent ne pas être cohérents entre les différentes sociétés et les différents sites. Enfin, la consommation d'énergie des réseaux est aussi importante que celle des centres de données, ce qui pose les mêmes problèmes de non-durabilité.

En ce qui concerne les ordinateurs personnels, nous avons assisté à une croissance sans précédent des appareils mobiles intelligents au cours de la dernière décennie. Cette croissance a été caractérisée par des progrès révolutionnaires en termes de puissance de calcul, de mémoire et de capacités réseau, permettant à ces périphériques de prendre en charge de toutes nouvelles catégories d'applications. De la réalité virtuelle et augmentée aux voitures autonomes, les opportunités semblent infinies. Cependant, beaucoup de ces applications nécessitent des latences très faibles (moins de 15 ms). Compte tenu de la forte augmentation du trafic dans le cloud et de l'incapacité des réseaux à soutenir cette croissance, il est impossible pour les clouds centralisés de prendre en charge ces applications. Outre les demandes de faible temps de latence, les applications mobiles génèrent une énorme quantité de données. Selon Cisco, d'ici 2021, 15% de toutes les données résideront dans des appareils mobiles, avec une grande chance d'être finalement transférées dans des centres de données centralisés. De plus, avec l'avènement de l'Internet des objets, la quantité de données générées atteindra 850 ZB par an, dont 90% sont éphémères (nécessitant un traitement, mais pas nécessairement le stockage). Traiter de telles quantités de données dans le cloud centralisé serait irréaliste, étant donné que l'augmentation du trafic serait 20 fois plus importante que prévu (21 ZB). Même si seul le résultat de ce traitement était stocké (10%), il dépasserait quand même le volume du trafic total généré par d'autres services. Afin de faire face à ces limitations, des solutions alternatives telles que le calcul dans l'Edge ou le Fog doivent être envisagées.

Edge et Fog

Edge et Fog s'appuient tous deux sur l'idée de la répartition des ressources de calcul et de stockage sur le réseau, plutôt que de les avoir dans quelques centres de données centralisés. Alors que l'edge se concentre sur l'introduction de ressources aussi proches que possible des utilisateurs, à la périphérie du réseau, le fog est plus générique et envisage une répartition des ressources dans un continuum entre la périphérie et le cœur du réseau. Ces concepts visent à remédier aux faiblesses du Cloud centralisé. Ils minimisent le temps de latence pour les utilisateurs, tout en réduisant considérablement le trafic propagé aux centres de données centraux du cœur du réseau. Cela ne prend pas uniquement en charge les applications en temps réel et IoT évoquées, mais offre également une répartition plus équitable des ressources, similaire à la structure initiale d'Internet. Cela aide l'infrastructure actuelle à mieux gérer l'augmentation du trafic attendue pour les services dans le cloud.

Le Fog et Edge sont récemment devenus des buzzwords au sein de la communauté IT. Cet intérêt s'est reflété de la même manière dans la recherche, en particulier au cours des cinq dernières années. Un nombre considérable d'études a porté sur des architectures et des algorithmes potentiels permettant de tenir ces promesses. Nombre d'entre eux se concentrent sur des applications spécifiques à un domaine, comme la santé, les véhicules autonomes et les villes intelligentes, tandis que d'autres se concentrent sur des aspects de gestion particuliers, tels que la planification, la migration, l'équilibrage de charge, etc.

Cependant, il convient de noter que la plupart des études sur le fog sont conceptuelle. Elles vont rarement jusqu'au développement d'un prototype. La matérialisation de ces idées en une plate-forme standard Edge / Fog à usage générique est toujours un problème ouvert, qui nécessite des recherches supplémentaires.

Une étape vers la normalisation consiste à traiter la gestion de l'infrastructure dans son ensemble, sans élaborer de solutions complètes pour des tâches spécifiques. Dans un premier temps, il est important de traiter les aspects architecturaux communs à toutes les tâches de gestion, tels que la répartition des calculs, l'évolutivité, les primitives de communication, la localité, les interfaces et la gestion de la qualité de service. Les décisions de conception sur ces aspects doivent toujours être conformes aux prémisses des architectures Edge / Fog concernant la réduction de la latence et de la génération de trafic. Une fois un cadre établi, la mise en œuvre d'une tâche de gestion particulière peut se concentrer sur les politiques spécifiques plutôt que sur l'architecture.

Objectifs

Dans cette thèse, nous abordons différentes étapes s'attaquant aux aspects architecturaux de la gestion du fog, en fournissant des blocs de base pouvant être utilisés dans la mise en œuvre de tâches de gestion spécifiques. Notre objectif est d'aborder ce problème dans deux directions : bottom-up et top-down. Plus spécifiquement, nos objectifs sont les suivants :

Bottom-up :

- Partant du principe du fog, notre objectif est d'identifier les contraintes les plus critiques pour une gestion d'infrastructure fog efficace.
- Sur la base de ces contraintes, nous visons à construire un réseau de recouvrement spécialisé pour un environnement fog, qui va servir de bloc de base pour les tâches de gestion de plus haut niveau.

Top-down :

- Afin de capturer les contraintes réelles gérées dans la couche de gestion, nous visons à déployer une application existante largement utilisée dans un environnement Fog.
- En l'absence d'une véritable application open-source native pour le Fog, nous visons à identifier les aspects architecturaux et les approches permettant d'adapter une application cloud afin de permettre son déploiement sur le Fog.
- Nous voulons savoir à quel point le déploiement dans un Fog peut être efficace pour une application non native pour le Fog et comment il peut être amélioré.

Intégration :

- Sur la base de notre réseau de recouvrement et de l'application déployée, nous visons à implémenter un exemple de tâche de gestion, telle que la découverte de service, qui s'appuie sur notre réseau et prend en charge le déploiement d'applications dans des scénarios utilisateur réels. L'objectif général est de fournir un prototype totalement intégré de gestion et de déploiement d'une application dans une infrastructure de fog.

Contributions

Organisées de la même manière que les objectifs, nos contributions sont les suivantes:

Bottom up :

- Nous affirmons que, pour respecter les principes du fog, une gestion efficace doit être simple, décentralisée, préserver la localité et réduire le trafic inutile, tout en offrant un service rapide et résistant aux pannes.
- Nous présentons Koala, un réseau de recouvrement qui fournit une implémentation de ces contraintes. Koala est entièrement décentralisé et adopte une structure simple et à plat, tout en tenant compte la topologie du fog. Il préserve la localité en utilisant différentes techniques, notamment via le routage et la découverte de nœuds proches. Koala minimise le trafic de gestion en supprimant toute maintenance périodique. Au lieu de cela, il inclut un mécanisme paresseux qui repose sur des informations de maintenance incluse dans le trafic des applications. Par conséquent, le niveau de maintenance est directement lié à l'utilisation.
- Nous fournissons une évaluation fondée sur la simulation de notre réseau de recouvrement dans diverses conditions, à l'aide de centaines de milliers de nœuds, et nous la comparons à un réseau logique classique.

Top-down :

- Nous discutons les contraintes d'applications qui doivent être déployées dans le fog sans être réimplémentées. Nous proposons les architectures à base de microservices comme un modèle prometteur pour de telles applications.
- Nous définissons une taxonomie des microservices fonction de leur état et de l'impact de leur réplication sur le déploiement de l'application. Cette taxonomie est essentielle pour décider comment et où les microservices doivent être déployés dans le fog.
- Nous considérons le déploiement dans le fog d'une application très utilisée à base de microservices, à savoir Sharelatex. Sur la base de notre taxonomie et de certains critères supplémentaires, tels que la criticité et la sensibilité à la latence, nous proposons un moyen de scinder et de répliquer les instances des microservices dans divers emplacements.
- Nous évaluons l'impact de ce déploiement sur certains cas d'utilisation d'applications et identifions plusieurs problèmes qui auraient pu être traités différemment si l'application avait été nativement conçue pour un déploiement dans le fog.
- Dans notre déploiement, le plus souvent, l'état des microservices est soit totalement répliqué dans différentes instances, soit scindé de manière à ce que chaque instance stocke un ensemble distinct d'objets. Nous observons que trouver le meilleur réplica ou localiser le bon fragment en fonction d'un objet est une exigence courante dans un tel déploiement et qu'il présente plusieurs avantages s'il est géré à l'extérieur de l'application.

Intégration :

- Au-delà de Koala, nous construisons un mécanisme de découverte de microservices qui gère les répliques et les fragments de données associées. Bien que la sélection des instances puisse être basée sur plusieurs critères, nous considérons la latence faible comme critère principal dans un environnement de type fog. Pour la localisation de fragments, nous introduisons une interface entre l’application et le mécanisme de découverte qui permet à ce dernier d’identifier les objets dans une requête et de rediriger la requête vers le fragment contenant cet objet.

- En plus de maintenir une correspondance entre les objets et les fragments, notre système effectue un suivi de l’emplacements des utilisateurs qui demandent un objet particulier. Si leurs emplacements sont éloignés de l’objet, notre système migre l’objet vers un autre emplacement, celui étant en moyenne plus proche des utilisateurs de cet objet.
- Nous intégrons notre mécanisme de découverte à ShareLatex dans un déploiement fog entièrement automatisé et montrons comment le système réagit à des cas d'utilisation réels.
- Enfin, nous revenons sur les leçons apprises non seulement pour le déploiement d’applications existantes dans le Fog, mais également pour la création native des futures applications destinées au Fog.

Résumé in English

The Cloud

As the need for computation power in the last decade has drastically increased, for many companies it has become unsustainable to host their own server infrastructure due to increased energy and management costs. Similarly, new companies were faced with high barriers for entering the market, because they required a large initial investment in order to be competitive. Therefore, owning on premise server infrastructures was soon replaced by moving computation to the Cloud. Currently, cloud computing has grown to become the standard deployment environment for most distributed applications. Based on the principles of utility computing, the Cloud provides a model for provisioning computing resources, infrastructure and services on demand similar to standard utilities such as water and electricity. The basic characteristics of cloud computing include: on-demand self-service, broad network access, resource pooling, elasticity and measurability. On-demand self-service implies that users can obtain or release resources as needed without prompting the provider. The obtained resources can be accessed via well-known network protocols supported by most of the devices. These resources are pooled between different tenants; as a user releases a resource, it becomes immediately available to other users. Elasticity implies that resources for an application can be automatically scaled up or down at runtime depending on the workload. Finally, the cloud uses a metering system for reporting the resource usage, which is leveraged in the pay-per-use business model.

Cloud numbers and limitations

Cloud services are one of the most rapidly growing markets, which is projected to keep a steady growth above 14% until 2022. According to a report by Cisco, by 2021, 94% of the overall computing will be processed in cloud datacenters. The global annual cloud traffic from 2016 to 2021 will increase 3.3-fold, and data stored in the Cloud will increase 4-fold .

While the continuous cloud growth seems far from plateauing, the effect of this growth on the current infrastructure needs to be examined. The end-to-end infrastructure comprises three main elements: the datacenters, the network (links, routers, switches), and the user machines or devices. Given that datacenters are owned by cloud companies directly benefiting from this growth, their reaction is likely to be more adequate. Indeed, Cisco's report foresees that hyperscale datacenters will dominate the landscape of public cloud by 2021. However, massive datacenters come at extremely high energy consumption. In 2030 consumption of datacenters will account for up to 13% of the global consumption, compared to 1% in 2010. Unless there is a profound shift to green energy, the growth of datacenters will not be sustainable.

Regarding networks, the response to cloud growth is less immediate. Different network segments are owned and managed by different actors and this creates several issues. Inadequate regulation and competitive dynamics between networking companies can result in sub-optimal infrastructures or configurations. Moreover, investments in network require a capillary intervention in infrastructure, which is expensive and has many technicalities (trenching, government permissions). This means that they

cannot be consistent among different companies and locations. Finally, the energy consumption of networking is as steep as that of datacenters, thus it suffers from the same unsustainability issues.
As regards user machines, in the last decade we have witnessed an unprecedented growth in smart mobile devices. This growth has been characterized by groundbreaking progress in computing power, memory and network capabilities, enabling these devices to support whole new categories of applications. From virtual and augmented reality to self-driving cars, the opportunities seem endless. However, many of these applications require very low latencies to operate (below 15 ms). Given the steep increase of cloud traffic and the inability of networks to sustain this growth, it is unattainable for centralized clouds to support these applications. In addition to the demands for low latency, mobile applications generate an enormous amount of data. According to Cisco, by 2021, 15% of all data will reside in mobile devices, with a high chance to move eventually to the datacenters. Moreover, with the advent of the Internet of Things, the amount of generated data will soar up to 850 ZB per year, where 90% of it is ephemeral (requiring processing, but not necessarily storing). Processing such amounts of data in the centralized cloud would be unrealistic, given that the increase of traffic would be 20 times more than predicted (21 ZB). Even if only the result of such processing was stored (10%), it would still surpass the amount of the overall traffic generated by other services. In order to deal with these limitations, alternative solutions such as edge or fog computing are to be considered.

Edge and Fog

Edge and Fog both rely on the idea of distributing computing and storage resources across the network, rather than having them in a few central datacenters. While Edge focuses on introducing resources as close as possible to the users, at the edge of the network, Fog is more generic and envisions a distribution of resources in the continuum between the edge and the core of the network. These concepts aim at addressing the weaknesses of the centralized Cloud. They minimize the latency to the users, while drastically reducing the traffic propagated to the central datacenters in the network core. This does not only support the aforementioned real-time and IoT applications, but also provides a fairer distribution of resources, similar to the initial structure of the Internet. This helps the current infrastructure to handle better the expected increased traffic for cloud services.
The premises of Fog and Edge have converted the two terms into a buzzword among technology enthusiasts. This interest has been reflected similarly in research, especially in the last 5 years. A considerable number of studies have focused on potential architectures and algorithms for delivering on these promises. Many of them focus on domain-specific applications such as healthcare, self-driving vehicles and smart cities, while others focus on particular management aspects, such as scheduling, migration, load balancing etc.
However, it is worth noticing that most of the Fog-related studies are conceptual and rarely come with a prototype. Materializing these ideas into a standard general-purpose Edge/Fog platform is still an ongoing process, which requires further research. A step toward standardization is to treat infrastructure management as a whole, without building up solutions for specific tasks from the ground up. At first, it is important to address architectural aspects that are common to all management tasks, such as task distribution, scalability, communication primitives, localization, interfaces, and Quality of Service (QoS) management. The design decisions on these aspects must still comply with the premises of Edge/Fog

architectures about minimizing latency and traffic generation. Once we establish a framework, the implementation of a particular management task can focus on policies rather than architecture.

Objectives

In this dissertation we go through different steps of addressing architectural aspects of fog management, by providing building blocks that can be used in the implementation of specific management tasks. We aim at tackling this problem in two directions, bottom-up and top-down. While in the bottom-up direction, we analyze management starting from the infrastructure, in the top-down one, we focus on the application perspective. More specifically, our objectives are the following:

Bottom-up:

- Starting from the premises of Fog, we aim at identifying the most critical requirements for an efficient infrastructure management
- Based on these requirements, we aim at building an overlay network specialized for a fog environment, which serves as a building block for higher-level management tasks.

Top-down:

- In order to capture real-life requirements to be handled in the management layer, we aim at deploying an existing widely-used application in a fog environment.
- In the absence of a real open-source fog-native application, we aim at identifying architectural aspects and approaches to adapt a Cloud application in order to enable its deployment over the Fog.
- We want to learn how effective fog deployment can be for a non-fog-native application and how it can be improved.

Integration:

- Based on our overlay network, and the deployed application, we aim at implementing an example of a management task, such as service discovery, which builds upon our overlay and supports application deployment in real-case user scenarios. The overall goal is to provide a fully integrated prototype of management and deployment of an application in a fog infrastructure.

Contributions

Organized similarly to the objectives, our contributions are the following:

Bottom up:

- We argue that in order to comply with fog principles, an efficient management needs to be uncomplicated, decentralized, preserve locality, and reduce unnecessary traffic, but yet deliver a fast service and be resilient to failure.
- We introduce Koala, an overlay network which provides an implementation of these requirements. Koala is fully decentralized and adopts a simple flat structure, yet it takes in consideration the datacenter-based topology of the fog. It provides locality-awareness using different techniques, including proximity routing and discovery of proximity nodes. Koala

minimizes management traffic by removing any maintenance routine. Instead, it includes a lazy mechanism which relies on piggybacking maintenance information on application traffic. Therefore, its maintenance is tied to its usage.
- We provide a simulation-based evaluation of our overlay network under various conditions, using hundreds of thousands of nodes, and compare it to a classical overlay.

Top-down:

- We argue about the requirements applications must comply with in order to enable their fog deployment without re-implementing them. We propose the microservice-based architecture as a potential implementation of those requirements (inline with the osmotic computing approach).
- We define a taxonomy of microservices based on their state and the impact their replication has on the application deployment. This taxonomy is critical for deciding how and where the microservices should be deployed in the fog.
- We consider the fog deployment of a well-known microservice application, namely Sharelatex. Based on our taxonomy and some additional criteria, such as criticality and usage, we propose a way to split and replicate the application services in various locations.
- We evaluate how this deployment impacts certain application use cases and identify several issues that could have been treated differently if the application was designed with fog deployment in mind.
- In our deployment, most commonly the state of microservices is either fully replicated in different instances, or sharded in such a way that each instance stores a distinct set of stored objects. We observe that finding the best replica or locating the right shard given an object, is a common requirement in such deployment and that there are several advantages if it is handled externally to the application.

Integration:

- On top of our Koala overlay, we build a microservice discovery mechanism which handles both replicas and shards. While replica selection can be based on several criteria, we showcase low-latency as a relevant criterion in the fog environment. For shard localization we introduce an interface between the application and the discovery mechanism which allows the latter one to identify objects in a request, and redirect the request to the shard containing that object.
- In addition to maintaining a mapping between objects and shards, our system keeps track of the locations of the users who request a particular object. In case their locations are distant from the object's shard, our system migrates the object to a different shard which location is on average closer to the users of that object.
- We integrate our discovery mechanism with ShareLatex in a fully automated fog deployment and show how the system reacts to real-life use cases.
- Finally, we provide some insights and some lessons learned not only for the fog deployment of existing applications, but also for building fog applications from scratch.