

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS-DE-LA-LOIRE - IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 648
Sciences Pour l'Ingénieur et le Numérique
Spécialité : *Informatique*

Par

Marie DELAVERGNE

Cheops, a service-mesh to geo-distribute micro-service applications at the Edge.

A focus on the replication collaboration

Thèse présentée et soutenue à l'IMT Atlantique, Amphithéâtre Besse, le 16 mars 2023

Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Thèse N° : 2023IMTA0347

Rapporteurs avant soutenance :

Noël DE PALMA	Professeur, Université Grenoble Alpes
Pierre SENS	Professeur, Sorbonne Université

Composition du Jury :

Président :	Thomas LEDOUX	Professeur, IMT Atlantique
Examineurs :	Sara BOUCHENAK	Professeure, INSA Lyon
	Ronan-Alexandre CHERRUEAU	Administrateur Cloud, Direction Générale des Finances Publiques
Dir. de thèse :	Adrien LEBRE	Professeur, IMT Atlantique

ACKNOWLEDGMENT



There are a lot of people to thank for my work in my thesis. A lot of really great persons, who taught me a lot, whether it was directly related to my work, or for transversal knowledge. At some point, I might not be able to describe how useful they were and how lucky I was to talk with them, in good enough terms, but I will try anyway.

First of all, I would like to thank my supervisors. Adrien Lebre, my “official” supervisor, was the one to pull me into this great adventure that is a PhD thesis. I saw firsthand how difficult it can be, and I would never have gotten into it with anyone else. I knew how dedicated he is with his other PhD students, and though it was not always easy, of course, he helped and supported me however he could, and pushed me whenever needed to give my best in my work. Ronan-Alexandre Cherrueau, who definitely was my supervisor even though it never was official, for the first year and a half. Thanks for your really good ideas, pedagogy, and support.

I would also like to thank everyone in my defense jury, Sara Bouchenak, Noël De Palma, Thomas Ledoux and Pierre Sens, first for their time and second for their great questions and remarks during my defense.

Then, all the people who participated in the project at some point. Matthieu Simonin and Javier Rojas Balderrama, for Openstackoïd and all the huge help in the beginning of my thesis. David Espinel Sarmiento, with his thesis which initiated the Cross collaboration, and great input on how it works, and a great partner in sea kayak. Karim Manaouil and Rodolfo, who participated in the Cheops meetings we had in spring 2021. Especially thanks to Karim with his help on Kubernetes and Cross and always had a great perspective on different things. My interns from spring 2021, Matthieu Juzdzewski and Arnaud Szymanek, who were a great help to understand Consul and how Cheops should be implemented, and were really great to supervise. Last, but definitely not least, Geo Johns Antony, for all his great knowledge and input, his patience, and the draft of the collaboration model. I wish him all the best for the end of his own thesis, even though I am going to be around to try and help.

Then comes the people from the STACK team. I will try to avoid thanking everyone by name, but some stand out. Anthony Simonet, who took me as his intern one day,



and was a great supervisor and a huge help. Johnathan Pastor, for his work on ROME, without which I would not have been in the team either, who was always a good friend and a really nice person, with insightful knowledge that helped a lot during all these years. Thuy Linh Nguyen and Jad Darrous, who were not only friends, but convinced me someday to pursue a PhD thesis. H           who was always a good friend too, and one of the nicest person I had the chance to meet. Maverick Chardet,           , Fatima-Zahra Bouj     also need to be thanked, and more recently, Antoine Omond and Baptiste Jonglez.

I am not going to mention every single one of my teachers, but I would like to thank them anyway. In particular, I would like to thank my university teachers from my first Licence in Universit   du Havre Fabrice Durand and V           Jay and of course all of my teachers in my computer science degrees from Universit   de Nantes, with a special mention to Florian Richoux, Fr             and Hala Skaf-Molli who really inspired me during the long road to PhD student.

Because a thesis would never happen without people from the administrative part, I would like to thank everyone who helped me to achieve mine. Thanks to Anne-Claire Binetruy and Catherine Fourny to help me with Inria and IMT Atlantique relations and being extra patient with my inability to deal with administration. Thanks to Michelle Dauv   for her work during my first year, and more to Delphine Turlier, who was so helpful and always available for my strange questions.

I need also to “big-up” my friends, from Universit   de Nantes, Charl    , Charles-     and Sylvain, but also Samuel, Dorian, Robin. Nathalie, from Universit   du Havre, who supported me those long years, from afar. The Garithos crew, especially Logan and Brandon, who took all my complaints these three years without complaining themselves. The PhD students discord (as the people in it, obviously) also has its own place here, both for moral support and help to solve problems, and coincidentally celebrated its 3rd birthday on the day of my defense. Happy Birthday, may you forever be a beacon for PhD students.

Finally, I am going to fall even more in the clich  s by thanking my family for their everlasting support throughout the years. In particular, I would like to say thank you to my love Florent, who is always my greatest source of support, inspiration and knowledge, and has been for almost twenty years.

TABLE OF CONTENTS



Glossary	vii
Acronyms	x
List of figures	xi
List of tables	xiii
 Introduction	 2
 I Depiction of the context	 7
1 Cloud Applications for beginners	9
1.1 What is the Cloud?	9
1.1.1 A short overview of the Cloud	9
1.1.2 Cloud Infrastructures	11
1.2 Cloud applications	12
1.3 Conclusion on the Cloud	13
2 What is the Edge?	15
2.1 Definition and purpose	15
2.1.1 Overview of the Edge	15
2.1.2 Intent	16
2.2 Hostility of the Edge	18
2.3 Conclusion on the Edge	19
3 Bringing Cloud applications to the Edge	20
3.1 Defining principles and expectations	20
3.2 Premises of solutions	23
3.3 Why current solutions do not fit all our expectations	25



3.3.1	In particular: the issue of geo-distributing OpenStack with a distributed data store	26
3.4	Conclusion on the context	30
II	State of the Art	31
4	Comparison points	33
5	Managing Edge applications on Edge infrastructures	36
5.1	Orchestration	36
5.1.1	Decentralized and Fault Tolerant Cloud Service Orchestration [119]	37
5.1.2	Liquid computing and Ligo [74]	40
5.1.3	HYDRA: Decentralized Location-aware Orchestration of Containerized Applications [76]	41
5.2	Control planes and other managements	43
5.2.1	OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures [49]	44
5.2.2	OpenYurt [121]	45
5.2.3	Re-designing Cloud Platforms for Massive Scale using P2P Architecture [122]	47
5.3	Conclusion on managing applications on Edge infrastructure	50
5.4	A service mesh at the Edge?	51
5.4.1	Istio [129]	53
5.4.2	Linkerd [130]	55
5.4.3	Conclusion on service meshes	57
6	How to make Edge applications natively	58
6.1	Towards Scalable Edge-Native Applications [51]	58
6.2	Highly-Available and Consistent Group Collaboration at the Edge with Colony [50]	62
6.3	Conclusion on ways to make an application at the Edge	64
7	Comparison and conclusion on the State of the Art	66
7.1	Comparison	66
7.2	Conclusion	67



III	Developing a solution to use native Cloud Applications at the Edge	69
8	An approach dedicated to geo-distribution	71
8.1	A service-mesh approach	71
8.2	Scope-Lang, a DSL to reify locations and collaborations of requests	75
8.3	What kind of collaborations do we need?	78
8.3.1	Forwarding for resource sharing	78
8.3.2	Forwarding for resource replication	79
8.3.3	Forwarding for cross	81
8.3.4	More (and more about) collaborations	82
8.4	Classification of resources dependencies	85
8.4.1	Creation patterns for replication operations	86
9	Cheops, our solution to push Cloud Applications to the Edge	89
9.1	Towards a full implementation of our approach in Cheops	89
9.1.1	Implementation of collaborations	90
9.1.2	Cheops architecture	94
9.2	A deeper dive into the replication	96
9.2.1	Replication model	96
9.2.2	CRUD execution workflow	97
9.2.3	Dealing with faults	99
9.3	Testing Cheops replication	100
9.3.1	Current technology stack	100
9.3.2	Experiments	101
	Conclusion	106
10	Discussion	107
10.1	Application version	108
10.2	Consistency	108
10.2.1	Consistency from the API	108
10.2.2	Better consistency	109
10.3	What can be improved in Cheops	109

TABLE OF CONTENTS



11 Towards a generalized control system: future work	111
11.1 Mid-term deliveries for Cheops	111
11.2 Ownership Types to prevent meaningless collaborations: the long haul . . .	111
12 Conclusion	116
Appendix	120
Publications	125
Bibliography	127
Résumé en français	148



GLOSSARY



Application Programming Interface An API is a software interface allowing other programs to interact with the software functionalities that are exposed by this interface. One of the goals is to keep the software internals hidden and expose only what is relevant for software communications.

Cache A cache is a component that stores data closer to its usage point so future requests on this data will be served faster. For example, your browser saves logos or images of website so it does not need to download them again next time you are navigating them.

Cloudlet A cloudlet is a small-scale data center that is mobile at the edge of the internet.

Content Delivery Network CDNs are a layer in the internet ecosystem. They consist in geo-distributed networks of data centers that delivers content (such as images, softwares, videos, streaming media, etc.) to end-users by being close to them. For example, Netflix or Youtube use CDN in most countries to deliver their videos faster.

Data center A data center is a physical facility hosting networked computing and storage components.

DevOps DevOps is a portmanteau word composed of (software) development and (IT) operations. DevOps are mainly a set of practices that combines those in order to produce faster better products for customers. Globally, it enhances the application lifecycle through methods used during the lifecycle phases, plan, develop, deliver and operate.

Disk array A disk array is a set of disk drives that compose a disk storage system.

Domain Name System The DNS is a naming system used to identify computers by a human friendly domain name by mapping it to IP addresses. For example, 8.8.8.8 is the primary DNS server for Google DNS.

Hypervisor A hypervisor is a software that can create and run virtual machines (virtualization of computer systems).

Internet of things The internet of things consists in physical devices that carry sensors, have processing capabilities, and different technologies, to connect to other devices



and computing systems, not necessarily on the internet. For examples, smart homes can be composed of different devices that compose the internet of things, like cameras, thermostats, etc.

Loosely coupled Loose coupling refers to a model where the components of the system are connected but not heavily dependent, so they are not affected by changes or failures of others.

Microservice The microservice architecture is an architectural style that composes an application as a collection of services that are loosely coupled, deployable independently, and execute each a small and coherent part of the business functions.

Peer-to-peer Peer-to-peer computing is a type of architecture for distributed applications to spread their workloads between peers of the system on which it is deployed upon. Peers all have the same privileges and same function abilities.

Router A router is a network device dedicated to forwarding data packets between networks. On the internet, they direct the traffic where it is needed.

Server A server is a *computer* designed to process requests, offer functionalities and deliver data to other, possibly multiple softwares or devices that are called clients.

Service A service consists in a set of functionalities, typically using one or a few different resources, which allows a complete business goal. For example, a billing service, a user management service, in a shopping application

Smart contract A smart contract is a class of computer protocol that allows, check and execute the actions required by the terms of a contract. The main idea is to have automated agreements to ensure the terms of smart contract terms depending of predetermined factors.

Switch A switch is a network device dedicated to forwarding data between the devices connected to it.

Transmission Control Protocol TCP is one of the main communication protocol, complemented by the IP (Internet Protocol). TCP is based on the establishment of a connection between the client and the server before data are sent to deliver reliable data streams.

Transport Layer Security TLS is a security protocol that provides secured and private communications in computer networks. It used for HTTPS, but also emails, instant messaging, etc.



User Datagram Protocol UDP is one of the communication protocol over IP network. Contrary to TCP, UDP does not require to establish a connection between the client and the server.

ACRONYMS



API Application Programming Interface

blob Binary Large Object

CDN Content Delivery Network

DC Data center

DNS Domain Name System

DSL Domain Specific Language

gRPC gRemote Procedure Calls

IaaS Infrastructure as a Service

IoT Internet of Things

OT Ownership Types

P2P Peer-to-Peer

PaaS Platform as a Service

PoC Proof of Concept

PoC Proof of Concept

PoP Point of Presence

SaaS Software as a Service

SPoF Single Point of Failure

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

LIST OF FIGURES



1.1	Different service models of the Cloud	10
1.2	A typical Cloud application	13
2.1	An overview of the Cloud and Edge (Source: Usenix - HotEdge'18)	16
2.2	Overview of The Smart Homes from [55]© 2020 IEEE	17
3.1	Intra and inter-service collaborations	24
3.2	Different types of solutions	25
3.3	Boot of a Debian VM in OpenStack	27
3.4	Boot of a VM using a remote blob	28
3.5	Booting a VM at Site 1 with a blob in Site 2 using a distributed data store (does not work)	29
5.1	Augmented Decentralized CloudLightning Architecture	38
5.2	OpenYurt architecture (Source: OpenYurt's Github).	46
5.3	Architecture of a typical service mesh, inspired from [30]	52
5.4	Multiple clusters on Istio	54
5.5	Multiple clusters on Linkerd	56
6.1	An example of a Colony topology (Source: [133]).	62
8.1	Microservices architecture of a Cloud application	72
8.2	Two instances of a Cloud application on two different sites	73
8.3	Load balancing principle	74
8.4	Service mesh <i>mon</i> for the monitoring of requests	74
8.5	scope-lang expressions σ and the function that resolves service instance from elements of the scope \mathcal{R}	76
8.6	Scope σ interpreted by the geo-distribution service mesh <i>geo</i> during the ex- ecution of the $s.e \xrightarrow{\sigma} t.h$ workflow in App_i . Reverse proxies perform requests forwarding based on the scope and the \mathcal{R} function.	76
8.7	Resource sharing by forwarding between instances	78



8.8	Replication by forwarding on multiple instances	79
8.9	The otherwise (;) operator.	83
8.10	The around operator build upon (;).	84
8.11	The different dependencies	85
8.12	Behaviors to observe following the dependencies	87
9.1	Sharing uses services instances from different sites	90
9.2	Replication executes an operation on multiple instances	92
9.3	Cross gives the illusion that multiple resources behave as a single one. . . .	93
9.4	Cheops architecture	94
9.5	Actual replicant model in Cheops code.	102
11.1	Boot of a VM attached to a flat network locally	112
11.2	Launching a VM with a network, using collaboration	113
11.3	Dangling pointer example.	114
11.4	Ownership types (in purple) to prevent invalid collaborations	114
12.1	Configuration of a service in Envoy in the first version of Cheops.	121
12.2	Workflow of a sharing request in the first version of Cheops.	122
12.3	Grid'5000 and the Renater network.	123

LIST OF TABLES



5.1	Comparison points on Edge infrastructure management. (1) Only available when activating node autonomy ¹	50
5.2	Comparison points on service meshes.	57
6.1	Application characteristics and corresponding applicable techniques to reduce load (Source: Wang et al. [51]).	60
6.2	Comparison points on making applications at the Edge.	64
7.1	Comparison points on the overall state of the art.	66
9.1	Experimentations on Kubernetes - replication.	102

Introduction



INTRODUCTION



A beginning is a very delicate time.

Princess Irulan, Dune

Since the term *Cloud* was coined in the 1990s² [2, 3], the Cloud Computing paradigm has become a pillar of computing mechanisms, offering solutions for businesses, scientists, individuals. It is now omnipresent, and a lot of companies (over 60% [4]) use Cloud workspaces, whether it is private, public, or hybrid.

Traditionally, except for on premises (private) Cloud, huge data centers (DCs) are built in key locations (e.g., in terms of energy cost) to serve users requests from all over the world, or at least, from large parts of the globe.

However, there is a growing need for low latency applications to be executed as close as possible to the *clients* (clients as consumers of the application, that do not need to be direct human users, but can also be applications related to Internet of things (IoT), smart cities, etc.). This is the new rising paradigm, called Edge Computing [5]. The main goal is to have of multiple micro and nano DCs at the edge of the network, closer to the clients [6]. For example, Points-of-Presence (PoPs) at the edge of the network, could be leveraged to get this geo-distributed (geographically distributed on the entire globe) infrastructure, close to the clients [7].

Initially, the activities I conducted on the topic focused on revising a resource management system such as OpenStack [8] to manage those specific Edge infrastructures. To benefit from the geo-distribution of these infrastructures, distributed systems at the Edge have to face high latencies (between sites that are far apart) and frequent disconnections inherent to wide-area networks (WAN) [5, 9].

In order to manage one application on such a widely geo-distributed infrastructure, we also need to consider scalability, locality of the resources, resiliency to disconnections and other site faults.

To deal with these challenges, one possible direction could be to build applications specifically for the Edge, with the challenges in mind [10, 11]. The Discovery Initiative [12]

2. the invention of the Cloud itself was earlier, around the 1950s/1960s [1].



followed different directions; when I began my work in the team, the main idea was to manage Edge infrastructures by revising a Cloud computing management system, (namely OpenStack), to allow it to run at the Edge.

The initial approach was about having the bulk of the application on the Cloud, where non-critical operations requests are treated, and handle the other requests (latency sensitive) on smaller, but more numerous and heterogeneous nodes deployed at the Edge [13]. In other words, instances of each critical service of an application must be deployed on Edge sites to fulfill the Edge objectives (mainly relative to the latency). Such a deployment of multiple instances is a problem because of the statefulness of services [14, 15], especially because in most real-life scenarios, services are stateful [16]. To clarify, splitting a stateless service is straightforward, and you mostly need to split an application into its different *microservices* and decide what can be distributed and what should stay centralized [15]. But splitting stateful services is a conundrum, as it requires to deal with synchronisation issues in a specific manner for each service of each application, with the necessity of knowing their inner functioning.

To help solving that puzzle, one of the research directions we followed is to address the resource sharing between services directly at the database level [17, 18, 19]. This solution consists in using a globally distributed database as a shared memory space for the services to use [20]. The assumption underneath is that developers can then write an application on top of them without thinking about distribution [21, 22]. This approach is nonetheless not straightforward as it requires to study how to introduce the geo-distribution into the application code to manage resources on such a database. This seems contradictory to the assumption, and will be explained in the *subsection 3.3.1 of Part I*. Here, we will just mention that it is a matter of execution context that leads to dedicated code.

We discovered this contradiction while using a distributed database, namely CockroachDB [23], for OpenStack [24, 25]. OpenStack is a huge system, with around 13M lines of code [26] and it is used primarily for the Cloud. Therefore, changing the code to manage resources in a geo-distributed database is not desired as some Cloud-native applications can be huge, and thus the necessary changes would be colossal.

Even worse, to manage the geo-distribution, not only it requires tremendous efforts, but it also requires to entangle the geo-distribution aspects into the business code (e.g., how to share states between *services* across multiple locations), which goes against the principle of separation of concerns. This principle widely adopted in the Cloud computing where a strict separation between development and operational (abbreviated as *DevOps*)



teams exists [27, 28]: Programmers focus on the development and support of the business logic of the application (i.e., the *services*), whereas *DevOps* are in charge of the execution of the application on the infrastructure (e.g., deployment, monitoring, scaling).

A principle which is enforced in the Cloud computing world by a concept called service mesh. A service mesh relies on the fact that application in the cloud computing are represented as a collection of *loosely coupled services* [29] to mitigate the operational complexity associated with modern applications [28] so that is decoupled from application code [30]. Thus, service meshes is the solution I studied to keep the geo-distribution concerns outside of the application code.

These problems motivated the work I defend in this manuscript. I propose a new approach that relies on the modularity of existing, service-based applications of the Cloud to leverage the geo-distributed infrastructures. This approach is generic to any of the applications that corresponds to:

Service-based The application needs to be modular and follow the rules of having different *services* managing different resources.

RESTful The services in the application must be REST compliant when communicating with each other.

Research topics and questions

Following this scope, this thesis aims at finding a generic way to use Cloud Applications at the Edge, with minimal to no impact on their original code.

Concretely, the research questions we address are:

- ❁ **Is it possible to use applications developed for the Cloud on Edge infrastructures without changing their code?**
- ❁ More specifically, can such an approach be used to manage a geo-distributed, Edge infrastructure, with an application designed to manage Cloud Infrastructures?
- ❁ And especially, since service meshes are designed to manage the communications between *services* of an application outside of its business code, can it be a solution to using Cloud applications on Edge infrastructures without changing their code?



Contributions

My work in this thesis brought three distinct contributions:

- The first contribution is mainly the theory of the approach, which was presented in Euro-Par 2021 [31], and a more specific contribution on the replication AMP workshop 2021 [32].
- The second contribution is the implementation of the approach, a Proof of Concept (**Proof of Concept**) called Cheops, presented at the Open Infrastructure Summit 2022 [33, 34] and as a short paper at ICSOC 2022 [35].
- To validate this prototype, as well as preliminary studies, I also contributed to the Enoslib proposal [36, 37], a library to help with experimentations on different infrastructures, which we will not discuss in this manuscript as it is out of scope.

To add more insight on my work during my thesis, while it has been funded by Inria, I worked also with people from Orange Labs through different projects in the Discovery initiative [12], such as [36, 25, 38, 7], on which my thesis is based. I began working in the team on OpenStack, and then the work was broaden to include Kubernetes [39], with perspectives directed on more applications to ensure the genericity of the solution. The second goal of working with OpenStack and Kubernetes was to try and answer the second research question, as those are two systems allowing to deploy applications on Cloud infrastructures. Understand these huge applications to know how to manipulate them was part of my work during this thesis.

Manuscript organization

This thesis manuscript is composed of four parts.

The first part presents the context and background of this thesis, and follows the Cloud (first chapter) to the Edge (second chapter) logic. The third chapter explains in detail the possibilities to have a Cloud application running on the Edge and paves the way to the following parts of the thesis.

The second part presents the state-of-the-art approaches to tackle the challenges at the Edge. It is divided in four chapters.

- First, I explain how I compared the different solutions to our requirements.



- Second, we see how applications can manage Edge infrastructures, and in particular how Cloud management applications can handle the shift to the Edge. This chapter has a section dedicated to existing service meshes to see if they fit our requirements and needs.
- Third, I give an overview of how it is possible to develop an Edge-native application.
- Finally, I conclude on what points are interesting to keep in mind for our solution and what is not appropriate.

The third part presents the approach envisioned in my thesis. In the first chapter, we discuss the overview of the approach in a theoretical manner. In the second chapter, we dive into the implementation of this approach.

The fourth part finally consists in three chapters, composed of a critique of the approach, a sketch of what is coming in the future for our approach, and the conclusion per say.

PART I

Depiction of the context





You still don't understand what you're dealing with, do you? The perfect organism. Its structural perfection is matched only by its hostility.

Ash, Alien

In this part, we present the context of bringing Cloud applications to the Edge without meddling in their code, the actors involved, and the difficulties implied.

In [Chapter 1](#), I first depict the existing and well known part in this problem, the Cloud computing and in more details, Cloud applications.

Then, in [Chapter 2](#), I state what is the new paradigm, Edge computing and what can be expected from Edge infrastructures.

Finally, in [Chapter 3](#), I develop an overview of what is needed for such infrastructures and why the existing solutions are not appropriate for the requirements presented.

CLOUD APPLICATIONS FOR BEGINNERS



In this chapter, we will explain how Cloud applications work, on what type of infrastructure they run, what is their purpose, what are the underlying assumptions for their proper functioning, and how we can manage a Cloud infrastructure with a Cloud application. This chapter is intended to be understandable by people with little to no background in the Cloud.

1.1 What is the Cloud?

Even if you do not know exactly what the Cloud is, you probably at least already heard about it before, somewhere, and have a vague idea of its purpose. A lot of the software you can use now propose offers to save your data somewhere *safer* than your own computer, namely “*the Cloud*”. We will now discuss what it is in more details.

1.1.1 A short overview of the Cloud

Cloud computing allows users to use computing resources (e.g., [servers](#), network, storage, etc.) on-demand without them having to manage these resources more than they need. The main goal is to be able to use resources in computing infrastructures, without the need to actually operate *physical* resources. This allows consumers (businesses, users, etc.) to avoid the cost, complexity and lack of flexibility of having to assemble and maintain these infrastructures. These computing resources are physically located in [data centers](#) (DC), sometimes huge (some of the largest exceed 1M square feet [40]), and pooled together to be spent by different users, according to their needs.

[Figure 1.1](#) presents an overview of the services available in the Cloud. Clients, such as web browsers, IoT devices, actual users, access and use the Cloud, which can be private (for one organization internally or externally), public (offered over the public internet), hybrid (composed from a public and a private part) or multiple (from different Cloud

providers), etc. Usually, the standard services models are presented as layers, from less to more abstracted resources:

Infrastructure as a Service (**IaaS**) offers low level resources such as physical **servers**, networks, Virtual Machines (VM), etc. as usable resources for the clients. As such, the clients have control over the deployed applications, networks resources, etc. on which they will work. Examples of IaaS are **Amazon EC2**, **Microsoft Azure**, **Google Compute Engine (GCE)**, **OVHcloud**, etc.

In Platform as a Service (**PaaS**), the clients are offered an environment, as a set of applications deployed, that will allow them to develop, run and manage applications. **Heroku**, or **Amazon Lambda** are examples of PaaS.

Software as a Service (**SaaS**) offers direct access to software. More and more desktop applications now have a Cloud version to be available for users as it would on their computer, but does not need the underlying requirements for the users points of access (for example, a user on its computer or smartphone does not need to have a large CPU, or storage), since the computing (or storage) will be done in the Cloud. As the most famous examples, we can cite **Google Workspace**, **Microsoft Office on the web** or **Netflix**.

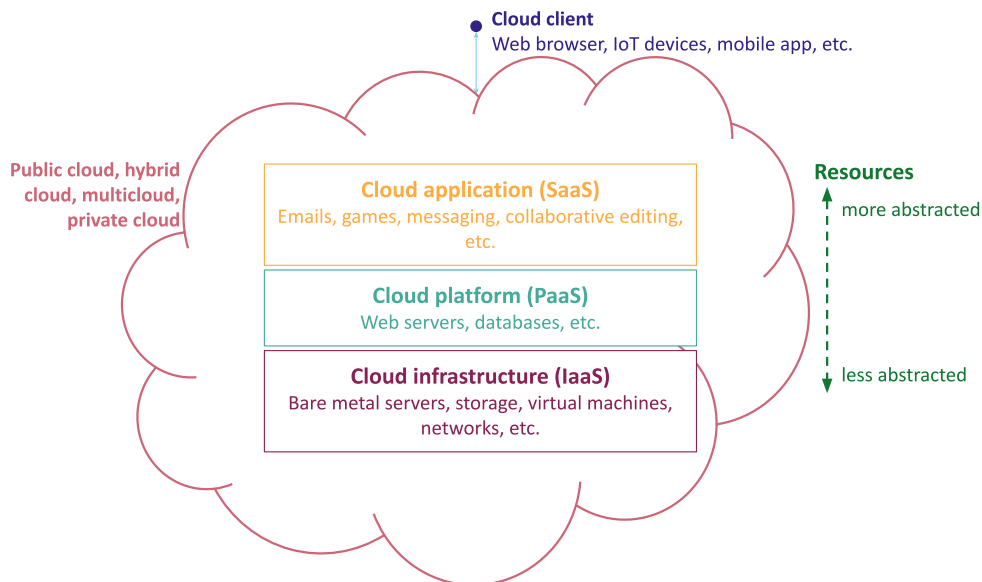


Figure 1.1 – Different service models of the Cloud

To be more formal, the NIST (National Institute of Standards and Technology) defined 5 principal characteristics for the Cloud computing model [41]:



On-demand self-service. A user can interact with service providers without the need for a human, to get computing resources, such as storage, [servers](#), network. We will talk more about these resources in the next [subsection 1.1.2](#).

Broad network access. These resources are delivered through the Internet network and can be accessed from heterogeneous devices (mobile phone, laptop, etc.).

Resource pooling. Cloud computing resources are pooled to serve multiple users at the same time (this is called multi-tenancy), and can be dynamically assigned depending on demand. Usually, the users have neither control nor knowledge of the exact location of the resources they are served. Sometimes, they may be able to specify the global location (such as country, region, [data center](#)).

Rapid elasticity. Resources can be dynamically provisioned, used and released to scale up or down automatically per user demand. They often can appear virtually unlimited to the users.

Measured service. The resources usage is monitored, which allows to optimize it. This usage can be measured, controlled and described to provide transparency for both the users and the provider.

1.1.2 Cloud Infrastructures

Whether we are speaking of large [data centers](#) or smaller, on premises, sets of hardware components, Cloud infrastructures building blocks are usually collocated and as homogeneous as possible. This is the hardware layer.

An abstraction layer virtualizing resources is often used to bring a better infrastructure logic to allow network and system administrators to manage these infrastructures more easily. This is the infrastructure layer that can be directly offered as a service (IaaS, as seen in [subsection 1.1.1](#)). Then come platform and applications layers (PaaS and SaaS).

Hardware, storage, network and virtualization compose the Cloud infrastructure.

Hardware includes [servers](#), as can one expect, but also [disk arrays](#), network components such as [routers](#), [switches](#), etc.

Virtualization is the component that links physical hardware from abstracted layers. Typically, a [hypervisor](#) abstracts the underlying hardware resources to link them logically and make them more easy to manage as pools of global resources in the infrastructure.

Storage is managed to back up data automatically or manually, ensure that older back-ups are automatically removed when they are no longer relevant, etc. It abstracts the physical storage through virtualization to offer Cloud storage solutions.

Network is composed of the physical network components on which virtual networks are created. Typically, virtual networks have different virtual resources that can be used, such as static/dynamic IPs, private or public sub-networks.

All the virtualized resources, including the network resources, are pooled as usable entities “independently” of the physical resources under, which means a user can exploit for example storage from two different parts of the same **data center** without even realizing it, as it was one single entity.

1.2 Cloud applications

Cloud applications, also called Cloud-native applications because they were developed specifically for the Cloud, are the software that users can access to execute services remotely, in *the Cloud*. More specifically, these applications run on top of two systems: client-side (for example, the users’ computer, or their browser) and server-side (which is contacted by the application on the client-side).

There are a lot of types of Cloud applications. But for most of them, they are a set of independent, **loosely coupled**, small **services**, connected with each other to form an entire, functional application.

In the rest of this thesis manuscript, we will use indiscriminately **microservice** [42] or **service** to define these services that are used to decouple functionalities in Cloud applications. It is important to define also here that though we talked about infrastructure computing resources before, from now on, we will use the word *resource* alone (unless it is obvious we talk about infrastructure resources) to describe values that can be manipulated by an application, and some can be stored in a data store, but it is not mandatory. These resources can also have side effects when manipulated.

Figure 1.2 presents a typical **microservice** based Cloud application. The blue rectangle around represents the entire Cloud application. A user (Andy), asks to **CREATE** a resource **a**. The calls are shown in **orange**, full arrows ; the responses in **orange**, dashed arrows. **ServiceA** represents the service called by Andy, which requires a sub-resource **foo** from **ServiceB**, to create the requested resource **a**. Notice that **ServiceA** effectively requests

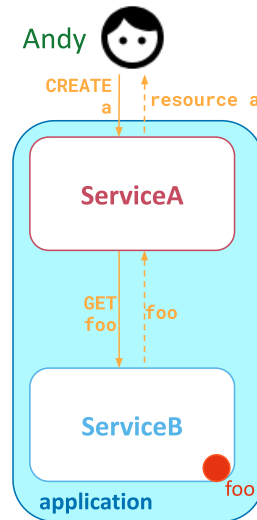


Figure 1.2 – A typical Cloud application

(GET) automatically, by itself, the sub-resource `foo` from `ServiceB`, without any more intervention from the user.

In a more practical example for this figure, imagine that Andy wants to create an account on an e-commerce website. `ServiceA` could be the service which creates and manages accounts, and `ServiceB` a service which creates and manages a randomly generated image as an avatar for the account. Andy asks an account creation, and `ServiceA` will execute this request, getting a random image from `ServiceB` to create the account. In this case, resource `a` would consist in different information on the account, and `foo` the image.

1.3 Conclusion on the Cloud

The Cloud is composed of hardware and software capabilities offered to users, who do not have to manage the underlying infrastructure or even have the compute capabilities. Users only require a device to connect to *the Cloud* thanks to the device they do have to get the resources they need to complete their tasks.

This comes with benefits such as scalability and flexibility of Cloud resources, availability of the offered services without having to own and maintain the required hardware. The Cloud is usually physically located in public or private data centers, that can be really huge and are typically not really close to the users, but rather placed where costs



for the infrastructure are low.

Cloud applications, running in the Cloud, are the ones that will be manipulated by users to achieve their goals in the Cloud, and are often composed of services that fulfill one functionality of the application and are connected together to achieve the entire application purpose.

We will now discuss what is the Edge computing paradigm, which comes from the Cloud computing one.

WHAT IS THE EDGE?



In this Chapter, we are going to see what is the Edge and how it differs from the Cloud. Then, we will discuss what are the challenges of running an application at the Edge, shedding more light on why it is difficult.

2.1 Definition and purpose

2.1.1 Overview of the Edge

The Edge is more recent and consisted originally in [Content Delivery Networks \(CDN\)](#), which [cached](#) and delivered large web content such as videos [\[43\]](#). Since then, the technology evolved to deliver more than static content [\[44\]](#).

The best way to envision the Edge as we consider it is to bring the Cloud (computation and storage) closer to the users. There are a lot of different views on what is the Edge, and how to define it.

Sometimes called Fog Computing, the part we are interested in is located in between the Cloud [data centers](#) and small devices that will collect data or just need some computation to be done [\[45\]](#). Some people only refer to the Edge as the devices that comprises the closest devices from the users, and Fog for the layer between this Edge and the Cloud [\[46\]](#), or Fog as the continuum between Edge and Cloud [\[47\]](#), or they use [cloudlets](#) as Fog [\[48\]](#). We refer to those as Edge devices (or sensors and controllers), and the [servers](#) were the computation will mostly take place as Edge [servers](#), or Edge [data centers](#), or for one set of collocated [servers](#), as an Edge site. Whatever the name, the *Edge data centers* will be the ones to make the hard work, considering the devices at the Edge of the network won't be able to carry it out, and will only collect data, sometimes pre-process them [\[49, 50, 51\]](#), and in some case, even those Edge sites will only do some parts, and the bulk of the tasks will be executed in the Cloud.

The *Edge data centers* we are considering are thus in charge of delivering Cloud ca-

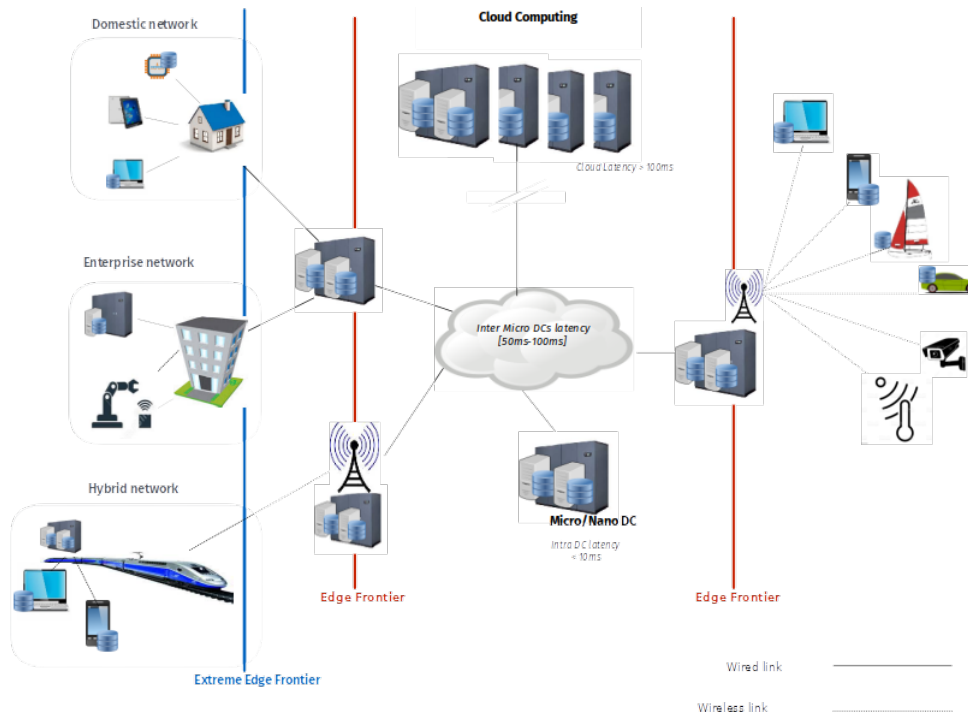


Figure 2.1 – An overview of the Cloud and Edge (Source: [Usenix - HotEdge'18](#))

pabilities to an Edge location (i.e., a region, a city, an business, an airport, etc.) and are composed of up to one hundred servers [31]. Figure 2.1 presents a view of Cloud data centers and our micro/nano-data centers we discuss about. They can be located anywhere, near the Edge devices, or in PoPs, on the Edge frontier. We consider that the latency from clients to the Edge data centers should be around or less than 10ms, when the one to Cloud data centers is between 50 to 100ms (or even more in worse cases, such as 150ms from Paris to Los Angeles [52]).

It is really important to understand that Cloud and Edge are fundamentally the same in terms of the logic of offering resources, storage, computing capabilities. One way to see it is: imagine a really small data center, and there are thousands of them, distributed all around the globe.

2.1.2 Intent

The most obvious purpose of Edge computing is to improve (lower) the latency between the location where a request is made and the place where it is executed. Indeed, since the distance is shorter between them, it is expected that the response time will improve, on

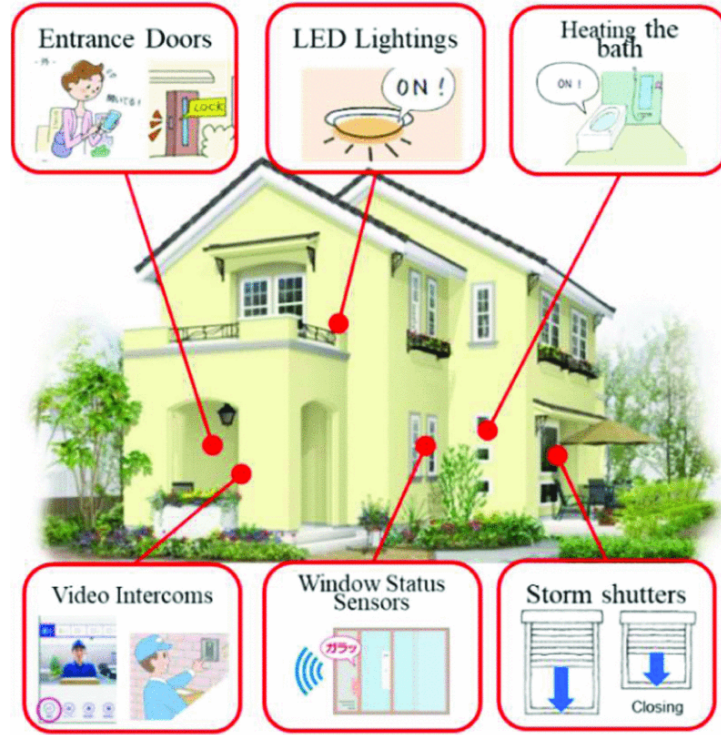


Figure 2.2 – Overview of The Smart Homes from [55]© 2020 IEEE

condition that the request is executed as quickly on an Edge location as it will be further away in the Cloud.

But by executing requests closer to the users, it is also expected to save a lot of bandwidth on the parts of the network between *the Cloud* (Cloud infrastructures) and the Edge. This implies a reduction in data transfer costs.

Another potential benefit is to try and use renewable energy to run the Edge infrastructures, by using farther sites when the closest location energy is not sufficient to cover the execution of requests [53], therefore increasing the sustainability of the Edge computing [54].

To explicit the aim of the Edge computing, it is better to state some of its potential applications. As mentioned, the main purpose of the Edge is to reduce the response time of *Cloud requests* and execute the requests closer to where they were made. The Internet of Things (IoT) is the most common example, with the need for scalability, and avoiding flooding the usual network with a lot of data.

And in particular, to give more concrete examples, Edge computing can be used, among other use cases, in agriculture, e.g., to improve yield [56, 57, 58]; in the oil industry, e.g.,



to monitor the transportation of the oil, for example [59, 60], in a house, e.g., to improve daily life or reduce energy costs [61, 62, 55] or for smart cities [63, 64]. Figure 2.2 shows examples on how to improve daily life with smart homes, such as to control via application or automatically through sensors different appliances in a house.

2.2 Hostility of the Edge

Though the Edge can be really powerful to avoid use of bandwidth (by going more local) and gives a lower latency for the applications requiring it, there are also some challenges that come with it.

It is a hostile and challenging environment [65, 66] for Cloud applications, with **latency** that come through the roof when different locations on opposite sides of the globe need to communicate with each other: the user is indeed closer to one Edge location, but maybe its request demands a resource that is far away and will need to be fetched. Moreover, overload can be generated on the network or in the overall application because of the scale and stability of the system can highly increase response time [67, 68].

Another challenge to be considered is the **heterogeneity** of resources on which the applications will run [69]. In our case, we do not really consider the Edge devices (e.g., IoT), but even with small **data centers** around the globe, it implies that probably different actors will be involved. And with different actors, or even often when we consider a single actor, the infrastructure (hardware, protocols and Operating Systems) of each site will probably differ from another [70, 68].

The **scale** of the infrastructure is also another challenge [71, 72]. Having small **data centers** around the globe close to the user implicitly says that there are going to be a lot of them; and that the applications running on it need to be highly scalable.

Alongside the two last challenges comes the difficulties of control and management of those sites, or put differently, the **operational constraints** [71, 68]. First, how to ensure the reliance of the equipment, network, and more generally the infrastructure environment in such a geo-distributed and at high scale. Second, how to have a global view of the entire infrastructure made from so much a large number of site and heterogeneous hardware. And third, how to ensure the security on this infrastructure, especially how to isolate workload while having a global view.

But probably the worst aspect of Edge computing is that **disconnections** between Edge locations are the norm rather than the exception [73, 74]. This is a huge problem



when considering pushing Cloud applications to the Edge. Cloud applications assume a high reliance of the infrastructure [75], so having parts of the pooled resources from the Edge infrastructure unavailable suddenly is definitely an issue [76].

2.3 Conclusion on the Edge

The Edge computing paradigm has been created to deal with more and more devices and applications requiring a low latency for the request they were making to the Cloud.

As several different definitions exist for the Edge, I need to reiterate here that in this manuscript, we refer as Edge, small **data centers** (up to a hundred servers), located near the users (less or around than 10ms latency), that will serve either as the main point of request executions, or as a relay for small requests, and redirect the larger requests to the Cloud.

There are a lot of applications for the Edge, but it comes with some inherent problems to tackle, such as heterogeneity, scale, operational constraints, and most important, **frequent disconnections** and the **latency** between different sites.

With these specifics in mind, we will now take an interest in how to put Cloud applications at the Edge.

BRINGING CLOUD APPLICATIONS TO THE EDGE



In this chapter, we state the principles and requirements identified in this thesis as necessary to bring existing Cloud applications to the Edge considering the challenges mentioned in the previous chapter.

The main point of this thesis is about bringing existing, functioning Cloud Applications to the Edge. Since some of these applications are huge, to push these applications from Cloud to Edge, we want to avoid changing their code as much as possible. These applications had some assumptions about the underlying infrastructure it would be deployed upon. These assumptions no longer make sense in the Edge context, with failures and disconnections happening regularly all over the infrastructure, as was just mentioned in the previous chapter.

3.1 Defining principles and expectations

In order to deal with the specific challenges stated in [Section 2.2](#), applications in Edge computing have to manage the geo-distribution of resources themselves [\[15\]](#).

In [\[77\]](#), the authors describe a distributed system as:

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

In a geographically distributed environment, we need to follow two major principles we introduced in [\[31\]](#):

Local-first : Minimize communications between sites and be able to deal with network partitioning issues by continuing to serve local requests at least.

Collaborative-then : Be able to take advantage of the different sites according to users' needs and infrastructure considerations.



The local-first principle has also been defined for *local-first software* in [78]; with seven ideals that should come with it: fast, multi-device, ability to function offline, collaboration between users, longevity, privacy, and user ownership and control. Moreover, for some applications such as social applications, the propagation of the content is often localized, so a lot of processing can be executed close to the point of origin [79].

A lot of Cloud applications were created not as monoliths, but by separating the code into collections of functionalities that apply one resource or a small set of resources, called services [80, 81], as mentioned in Section 1.2. Some approaches to use Cloud applications at the Edge use this to deploy different services/functionalities on different sites, or different instances of the same service on different servers or sites, for balancing purposes [46, 82, 83].

From my point of view, the easiest way to ensure the local-first principle is by having an entire instance of the application on each site, which coincides with the description of distributed systems above (autonomous computing elements). But autonomous instances alone do not provide a way to ensure the *collaborative-then* principle. In order to share diverse resources and enhance the capabilities of the geo-distributed cloud application, an instance on one site should be able to collaborate with other instances on other sites if need be [84].

With the collaborations between all the instances of every sites, we can get a single coherent system. This view of a single coherent system I am aiming at and described in the definition of a distributed system, is also comparable to the Single System Image (SSI), which is composed of physical or logical mechanisms to give the illusion that a set of distributed, heterogeneous resources (hardware) forms a unique and shared computing resource [85].

With those principles, we can have a geo-distributed application running at the Edge. But these principles are not enough to specifically bring Cloud applications to the Edge, with its inherent constraints. Thus, several requirements I deem necessary in this context of bringing Cloud applications to the Edge follow, with some of them joining the local-first software ideals.

Non-intrusive: One thing we do not want to do is change the code of the application to geo-distribute it. This is not mandatory for the Edge, but since it is a gain in time, it is one of the roots of my approach. The main reason is that some Cloud applications are huge and thus changing their code is tedious and afterwards difficult to maintain.



Generic: This is not a requirement for the Edge per-se, but in order to be largely used, the bulk of the effort to push an existing application to the Edge should be (almost) ready to be used by developers for their applications, with a minimum production from their parts to plug to the solution if necessary.

Tolerance to network partition: This is the criteria of being robust to network partition. Since the Edge infrastructure is globally distributed, an application deployed at the Edge must reckon with the challenges coming with a wide-area network [86], one of them being frequent disconnections and network partition. It is different from the local-first principle because this is not the only way to envision tolerance to network partition, such as replication and distribution of services.

Dynamic placement of requests execution: The location of requests execution should be handled dynamically. In the context of the Edge, it is important that requests can be executed dynamically as the Edge itself is pretty dynamic [87], with nodes connecting and disconnecting often. By default, a static service composition establishes the collaboration between services inside one instance permanently [88]. It presents great advantages for the developer. Modularity and static composition anticipates the services interface, behavior and location. Therefore, it guarantees the correctness of services' invocations and proper execution of the application [89].

In OpenStack, for instance, the compute service is configured, once and for all, to always request the image service in the same DC. Hence, the compute does not mess with an unacquainted service. However, these guarantees comes with the cost of an unyielding application [88, 90] that prevents on-the-fly collaborations.

On-demand execution location: The location of execution of requests should be chosen per request by the users of the application for several reasons [91].

First, the need for privacy and trust is really important in the context of the Cloud, and equally important in the Edge. To ensure that the users privacy is respected, it is always easier to let them choose where their resources will be located and manipulated, which sites they trust.

Second, to avoid overloading each node with resources, it is appropriate to have public resources scattered across the infrastructure and usable at will, only when needed. In this regard, it is intimately linked to the collaborative-then principle.

This requirement is crucial, because it is always possible to add a layer on top that would choose the location of requests execution either by learning where the users



usually execute their requests or depending on nodes availability and/or latency, or using the information on the nodes uptime for reliability [92, 93]. This layer could be added if the need arose to avoid lengthy and redundant information in requests or if a more transparent definition is required, e.g., for simplicity. But it would be more difficult to give the users the choice of defining their needs finely, on-demand, rather than automatize everything from the beginning.

Decentralized: In the goal of coping with the network faults, a centralized approach means some of the logic of the application will not be able to be executed in case of a disconnection (it is a Single Point of Failure).

A centralized approach is also a bottleneck, as all requests that need to be handled by a central authority will need to go through the same site, and goes against the goal of lowered latency of the Edge, as requests will need to go further most of the time (breaking our local-first principle as well).

Peer-to-peer (P2P): This properties derives from other aforementioned requirements (local-first, tolerance to network partitions, decentralized), but adds the necessity for equally privileged instances of the applications; and it goes along by default with having autonomous instances of an application everywhere. Moreover, as the main challenges of the Edge are frequent disconnections, latency and distributed resources between nodes of the infrastructure, it makes sense to check if the adopted solution can be considered as P2P approach as these are the challenges that a peer-to-peer system deals with automatically [94].

3.2 Premises of solutions

While the local-first principle can be ensured single-handedly by having an entire instance of an application on each site, it automatically prevents the use of resources between the sites if the application was not designed for this.

Thus, with such a deployment, we have isolated applications that only work with their own resources and we do not leverage the overall Edge infrastructure. Moreover, sharing resources between those different instances makes sense in the context of the Edge, where the mobility of the users and the locality of execution is crucial [68, 95]. Accessing a resource that is on another instance of the application requires additional pieces of code that is not really dedicated to the application business, and worse, that only has this particular function.

A way to share resources between multiple instances of an application is to leverage both the service composition and the fact that multiple instances of an application share the same services, that thus expose the same [Application Programming Interface \(API\)](#). This is simple inter-service communication (ISC) that goes from one site to another, as it can be used for load-balancing purposes. Unfortunately, though this type of collaboration is easy to realize, it does not provide a global view that would be expected as one single, coherent system.

To achieve this view, such as a SSI we mentioned in the previous [Section 3.1](#), we need to be able to manage resources between different instances of the same service. To allow this, each service must be extended with some code that allows more collaboration. We refer to this type of collaboration as intra-service collaboration, because it is used within the logic of one service, between multiple instances of the same service. The problem that goes with this dedicated code is that it goes beyond the initial purpose of the service and thus includes collaboration code in the business code of the service. Although this dedicated means can benefit from message passing middlewares or distributed databases internally, it always requires invasive and complex code [96]. This is a well-known, challenging, and error-prone task for developers of the application [97, 98, 99], who prefer not to take resource sharing into account most of the time. But the other side is that these sorts of collaborations allow a global vision of resources managed by the applications, because users are thus be able to retrieve any resource from anywhere in the system/infrastructure.

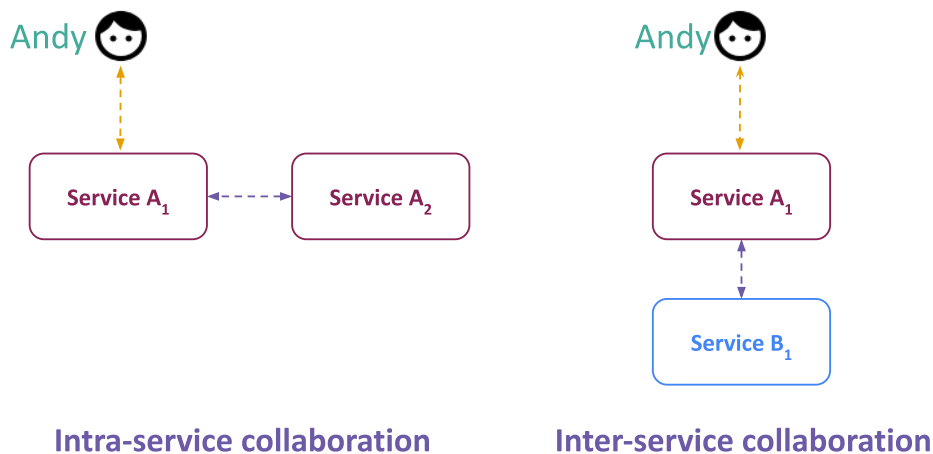


Figure 3.1 – Intra and inter-service collaborations

So we have two types of collaborations between services: intra and inter. The former implements the collaborations of different instances of the same service; the latter the col-



laborations between different services and, in our case, especially from different services across application instances or sites. Both types of collaborations are presented in Figure 3.1. In **intra-service collaborations**, we can see that the user interacts with one service, and there is *some kind* of collaboration between the two instances of the same service to fulfill the request. For example, getting some information from whichever database the other instance of the service is connected to (assuming they are not connected to the same database, as they would on two different autonomous sites). In **inter-service collaborations**, the collaboration is made between two different services. In two different instances of an application, it could be possible to execute this collaboration between two different instances of two different services. Both those types of collaborations ensure a single coherent system that leverages the entire infrastructure and every resources available.

3.3 Why current solutions do not fit all our expectations

To allow these types of collaborations, while having a single coherent system, three types of solutions exist in the litterature.

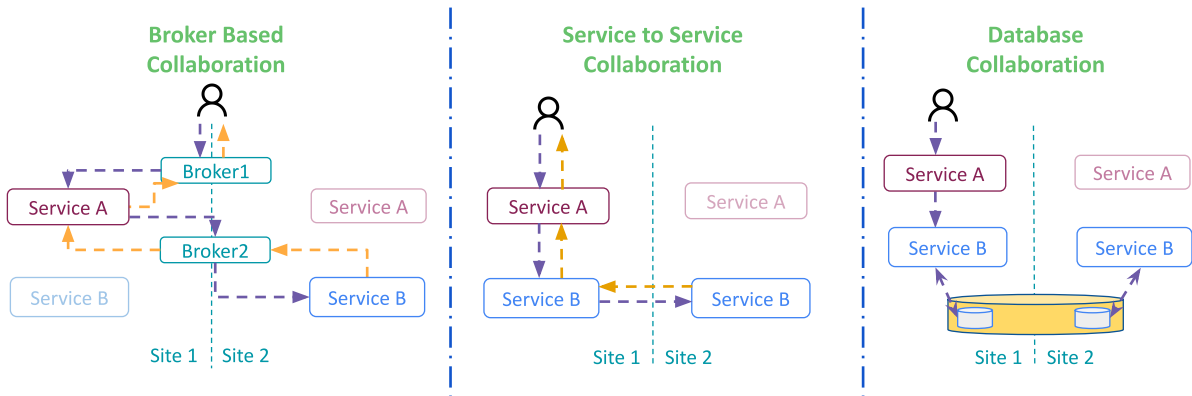


Figure 3.2 – Different types of solutions

Figure 3.2 presents these three approaches. The first is a broker based solution, which allows for inter-service collaborations. The broker approach allows externalization of collaboration aspects away from the business code. However, this approach requires a lot of development effort for an application. In addition to requiring a broker stub for each service, each broker should implement once again a lot of what the original functionalities



of the service (because the broker needs to offer the same capabilities as the underlying service in a geo-distributed and transparent manner). The alternative is having a generic broker to all services and applications, which is difficult to envision because of the differences between them.

Another interesting approach is service-to-service collaborations, which is an intra-service collaboration. As discussed before, this type of collaboration requires dedicated code in the service, which is not related to its core logic.

Finally, the third approach is the one we explored previously to this PhD [17, 24, 25], using databases or data stores, as the current accepted research direction for developing geo-distributed applications consists in using globally distributed data stores [100]. Distributed data stores emulate a shared memory space among the different instances of the application to make the development of geo-distributed application easier [21].

I will now present more thoroughly this approach, as we discovered by using it why the inherent problems to adopt it for an existing Cloud application.

3.3.1 In particular: the issue of geo-distributing OpenStack with a distributed data store

Most of this part comes from our Euro-Par paper [31], and is also discussed in our Research Report from 2020 [96], both written with Ronan-Alexandre Cherrueau and Adrien Lebre, and the second also with Javier Rojas Balderrama and Matthieu Simonin. To explain the issues with this approach, I will use OpenStack as an example, since it is by trying to use it at this Edge we identified these issues, and previous studies based on a shared database already paved the way for OpenStack [17, 101]. OpenStack is a resource management application to operate one DC. It is responsible for booting Virtual Machines (VMs), assigning VMs in networks, storing operating system images, administrating users, or any operation related to the management of a DC. It is thus both an Cloud application and an application to manage Clouds. It is a good example of what we want to geo-distribute, as OpenStack is around 13M lines of code [26], so a huge application we do not want to change.

A complex but modular application.

Similarly to other Cloud applications, OpenStack follows a modular design with many services. It has been developed to run natively on one single [data center](#). Some efforts have

been made along its development to make it run on multiple clusters, even at large scale. For example, some involved to allow service-to-service collaboration for specific services (like Keystone [102], Glance [103]), but it requires a lot of efforts for all services to add the features and to maintain them. The compute service for example manages the compute resources of a DC to boot VMs. The image service controls operating system blobs like Debian. Figure 3.3 depicts this modular design in the context of a boot of a VM. The user starts by addressing a boot request to the compute service of the DC (Step 1). The

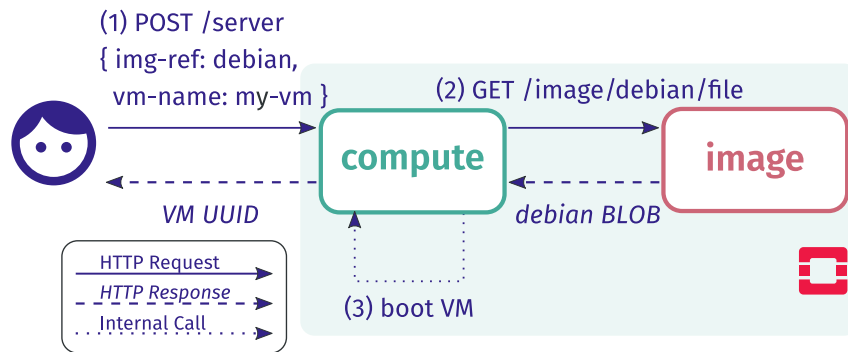


Figure 3.3 – Boot of a Debian VM in OpenStack

compute service handles the request and contacts the image service to get the Debian blob in return (Step 2). Finally, the compute does a bunch of internal calls, such as schedule VM, setup the network, mount the drive with the blob, before booting the new VM onto one of its compute nodes (Step 3)¹

Geo-distributing Openstack.

Following the local-first and collaborative-then principles implies two important considerations for OpenStack. First, each DC should behave like a usual cloud infrastructure where users can make requests and use resources belonging to one site without any external communication to other sites. Second, users should be able to manipulate resources between DCs if needed [95]. For instance, Figure 3.4 illustrates an hypothetical sharing with the “boot of a VM at one DC using the Debian image available in a second one” scenario.

To provide this resource sharing between Site 1 and Site 2, the image service has to implement an additional dedicated means (Step 2b). Moreover, it should be configurable

1. For clarity, the boot workflow is simplified here. In a real OpenStack deployment, the boot also requires at least the network and identity service. Many other service may also be involved. See <https://www.openstack.org/software/>. Accessed 2022-10-02

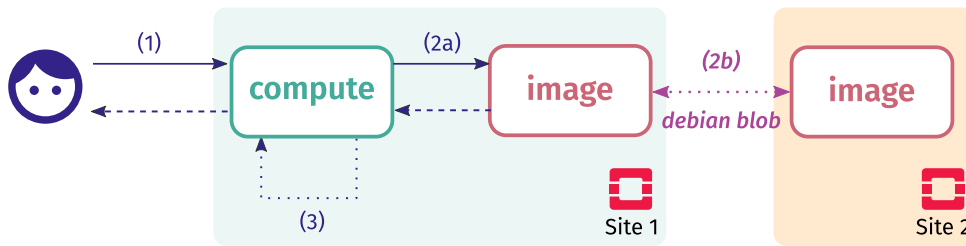


Figure 3.4 – Boot of a VM using a remote blob

Listing 3.1 – Retrieval of a blob in the image service

```

1 @app.get('/image/{name}/file')
2 def get_image(name: String) -> blob:
3     # Lookup the image path in the data store:
4     # `path = proto://path/debian.qcow`
5     path = ds.query(f'''SELECT path FROM images WHERE id IS "{name}";''')
6
7     # Read path to get the image blob
8     image_blob = image_collection.get(path)
9     return image_blob

```

as it might be relevant to replicate the resource if the sharing is supposed to be done multiple times over a WAN link. Implementing such a mechanism is a tedious task for programmers of the application, who might prefer to rely on a distributed data store [20].

Problem: distributed data store tangles the geo-distribution concern.

The OpenStack image service team studied several solutions to implement the “booting a VM at Site 1 that benefits from images in Site 2” scenario. All are based on a distributed data store that provides resource sharing between multiple image services: a pull mode where Site 1 instance gets blobs from Site 2 using a message passing middleware, a system that replicates blobs around instances using a shared database, etc. [103] The bottom line is that they all require to *entangle* the geo-distribution concern within the logic of the application. This can be illustrated by the code that retrieves a blob when a request is issued on the image service (code at Step 2 from Figure 3.4).

Listing 3.1 gives a coarse-grained description of that code. It first queries the data store to find the path of the blob (l. 5). It then retrieves that blob in the `image_collection` and returns it to the caller using the `get` method (l. 6–8). Particularly, this method resolves the protocol of the `path` and calls the proper library to get the image. Most of the time, that path refers to a blob address on the local disk (e.g., `file:///path/debian.qcow`). In

such a case, the method `image_collection.get` relies on the local `open` python function to get the blob.

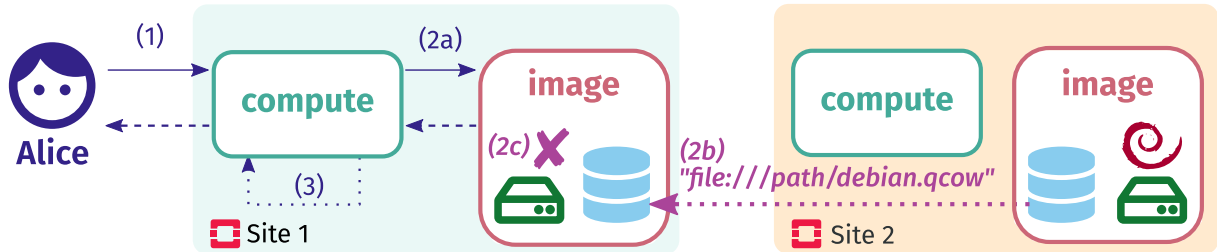


Figure 3.5 – Booting a VM at Site 1 with a blob in Site 2 using a distributed data store (does not work)

The code executes properly as long as only one OpenStack is involved. But things go wrong when multiple are unified through a data store. Figure 3.5 presents how it is a problem. If Listing 3.1 remains unchanged, then the sole difference in the workflow of “booting a VM at Site 1 using an image in Site 2” is the distributed data store that federates all image paths (including those in Site 2). Unfortunately, because Site 2 hosts the Debian image, the file path of that image returned at Step 2b is local to Site 2 and *does not exist* on the disk of Site 1. An *error* results in the `image_collection.get` (2c).

The execution of the method `image_collection.get` takes place in a specific environment called its *execution context*. This context contains explicit data such as the method parameters. In our case, the image `path` found from the data store. It also contains implicit assumptions made by the programmer: “A path with the `file:` prototype must refer to an image stored on the local disk”. Alas, such kind of assumptions are wrong with a distributed data store. The code has to be fixed. For this scenario of “booting a VM at Site 1 using an image in Site 2”, it means changing the `image_collection.get` method in order to allow the access of the disk of Site 2 from Site 1. More generally, a distributed data store constrains programmers to take the distribution into account (and the struggle to achieved it) in the application. And besides this entanglement, a distributed data store also strongly limits the collaborative-then principle. In the code above, there is no way to specify whether a particular blob should be replicated or not, for example.

The geo-distribution thus must be handled outside of the business logic and in a fine-grained manner due to its complexity, along with the requirements stated before.



3.4 Conclusion on the context

As we discussed throughout this part (Part I), Cloud applications were not designed to face the Edge challenges, especially the disconnections and latency problems inherent to this paradigm.

As we want to bring Cloud applications to the Edge, and considering that some are huge, we want to avoid changing their original code as much as possible and find a generic solution that can be used for a lot of applications.

This is why I defined these principles for a possible solution:

- **local-first** to work autonomously
- **collaborative-then** to leverage the infrastructure
- **non-intrusive** regarding the code of the application
- **generic** to a lot of existing applications
- **tolerance to network partition** for the Edge disconnections problems
- **dynamic placement** of requests to cope with the dynamism of the Edge infrastructure
- placement **decided by the users**, per request, when needed
- **decentralized** to avoid single points of failure complications
- **P2P** to match the aforementioned principles: local-first, tolerance to network partitions, decentralized

As I was defining the principles required for the solution I want to answer my research question of bringing Cloud applications to the Edge, I specified the logic of a solution for the collaborations between different instances of the exact same application deployed on each location while keeping a single coherent system.

We have overviewed why different solutions for collaborations, namely *non-generic brokers*, *service-to-service/federation*, and *distributed databases*, are not generic and imply to code the collaborations for each service (broker, service-to-service), or entangle non-business information into the code (database), which automatically goes against the non-intrusive principle.

In the next part (Part II), we will observe how the state-of-the-art solutions envision their own different ways to bring Cloud applications to the Edge, as well as taking a peek on how Edge applications can be developed to check if we can learn ideas on how to build this bridge between the two computing paradigms.

PART II

Using applications at the Edge: a state of the art





We got tactical smart missiles, phased plasma pulse rifles, RPGs, we got sonic electronic ball breakers! We got nukes, we got knives, sharp sticks...

Private Hudson, Aliens

In this part, we will compare academic and industry approaches to put an application at the Edge. We have seen in the previous part ([Section 3.3](#)) how approaches to deal with the collaborations to provide a single coherent system are not satisfying. Now, we will investigate how other approaches can be made without necessarily having to deal with the collaborations, by pushing applications to the Edge in different manners.

The goal of this part is mainly to study interesting solutions in the literature to first, obviously, avoid reinventing the wheel. This will be ensured by the comparison of our requirements to these approaches; if they fit my requirements, it is not necessary to make it from scratch. Second, even if some requirements are not enforced, parts of the approaches can be interesting to build my own approach, and thus answer the questions raised. Finally, it can be useful to give insights of functioning ideas that do not fit our requirements to be able to question them.

Because I wanted my solution to be generic to a lot of Cloud applications, there is a broad spectrum of approaches that can be of interest to us. As a smaller goal, I tried to focus on the most interesting approaches regarding our requirements and/or the representativity of the solution for each section.

First, I present in [Chapter 4](#) the comparison points based on the requirements we defined in [Section 3.1](#).

[Chapter 5](#) presents ways to manage the Edge infrastructure so we can deploy applications on it. In particular, [Section 5.4](#) presents service meshes for the Cloud to explore the possibility of using them at the Edge.

Then, I present in [Chapter 6](#) how it is possible to make Edge-native applications.

Finally, [Chapter 7](#) concludes on the overall comparison on papers.

COMPARISON POINTS



This chapter presents the different categories on which I will compare different approaches.

Because the comparison points are not entirely binary, the path I chose to compare the solutions is to attribute clouds (☁), from one to three. As a disclaimer, I tried to define the attribution of clouds as decisive as possible, but for some categories it is difficult to be unquestionable and some attributions might seem to be unfair for a specific approach. It is important to note that this comparison is not a grading operation and a difference from one cloud in the attribution does not really matter because we are not saying that an approach is better than another, but simply I am comparing them to my requirements. **If the category is entirely out of scope for a specific solution, such as if it was not considered at all, no clouds will be attributed.** I will now present, for each category, how I will attribute clouds.

Non-intrusive (NF) As mentioned, one thing we do not want to do is change the code of the application to geo-distribute a Cloud application.

Three clouds correspond to no touching at all; two clouds if the approach needed to change some of the vanilla code, and one cloud for touching the coding without changing its fundamental logic.

Generic For our solution, we chose to be as generic as possible.

Three clouds are given for approaches that would function on a large set of applications; two clouds when it is generic for applications that would be able to run on top of the approach (if it is available for Kubernetes-based solutions for example); and one cloud if it is only functioning for applications from the same development team, that share for example the same specific [API](#).

Local-first (LF) The local-first principle, explained in [Section 3.1](#), correspond to the ability of the solution to use at minimum the local site to serve requests in case of a network partition.



Three clouds were attributed if the application is entirely available locally; two if most of it is available at all time (e.g., an effort has been put for the most common requests); and one cloud if at least some part of the business logic is available on any site.

Collaborative-then (CT) Autonomous instances of an application should be able to collaborate to leverage the entire infrastructure and resources.

Three clouds are given for approaches that allow different sites to collaborate for any kind of resources; two when it is only a subset of resources; one when only some resources can be shared.

Network partition (NP) This is the criteria of being robust to network partition. It is separated from the local-first principle because this is not the only way to envision tolerance to network partition.

Three clouds are awarded for solutions that allows sites to work autonomously and/or merge the changes as required after; two clouds if resources are available in a read-only mode or in other way to get them and one cloud if network partition is tolerated only if another site is cut from the network or only in some specific cases.

Dynamic The location of execution of requests should be handled dynamically and not statically in a configuration file.

Three clouds are given for an approach in which the location is selected dynamically; two clouds if the information of sites locations is given in a configuration file but used dynamically afterwards and I did not find in the literature any approach that would define or justify one cloud, so no “one cloud” will be attributed on this point.

On-demand The location of execution of requests should be chosen per request by the users of the application for different reasons mentioned in [Section 3.1](#).

Three clouds were granted if users can choose requests execution location at fine grain while building their requests; two clouds if the user can define this location per request, one if they need to define the location globally and/or statically (and thus, also not dynamic), or if they do not chose the location, but is is decided by the application.

Decentralized The approach we envision must be decentralized to avoid losing some to all functionalities of the application. Usually, approaches that are not fully decentralized will not fulfill the local-first principle and thus, typically cannot be tolerant



to network partitions.

Three clouds were given for decentralized approaches; two if the approach is partially decentralized, but leverage the Edge-to-Cloud continuum to execute only some part of the workload in the Cloud; and one cloud if the main part of the application is centralized in the Cloud and the Edge only execute the rest.

Peer-to Peer (P2P) This is about equality of privileges on the different sites of the infrastructures.

Three clouds were given if all instances of the applications have the same privileges. Two if all the sites are equals but some sites are more *equal* than others, as George Orwell would have put it (i.e., if some sites do not have the exact same privileges as others). One cloud for approaches where most of the decisions are made on a few nodes and the bulk of the nodes are mostly producers/workers.

For every solution we will observe in this part, I will give an overview of the approach, the clouds attributed for each of these points of comparison and a short conclusion on the solution, with information on what was relevant exactly for me.

MANAGING EDGE APPLICATIONS ON EDGE INFRASTRUCTURES



In this part, we will focus on ways to leverage an Edge or Cloud/Edge infrastructure with virtualized resources on which we can deploy applications. To understand the importance of checking what was done for managing the Edge infrastructure, it is crucial to grasp these two points: (i) in the Cloud, infrastructure are managed by Cloud applications (offering *IaaS*), (ii) if we can push these Cloud applications to the Edge themselves, we would have Edge infrastructure managers that can be used to deploy applications following our requirements.

As our main goal is to push Cloud applications to the Edge, having management applications for the Edge *for free* is definitely worth investigating.

We will first investigate specifically applications able to orchestrate services, and after, we will see more general approaches.

5.1 Orchestration

Cloud Orchestration applications have the ability to automatically configure, coordinate and manage a Cloud system. They manage the life-cycle of services and provide users with the required services for their workload. They also monitor the infrastructure and services to adapt the system according to the requirements [104].

[105] is a survey on Fog orchestration, which corresponds mostly to what we define as Edge. The authors determined that among the main goals of the orchestration in the studied literature, resource management is the most cited, followed by guarantee of Service Level Agreements (what standards of services will be offered) and service life cycle management. Both services and applications are the most orchestrated entities (respectively in 36% and 34% of the studied literature), followed by tasks. The topology of the approaches in their corpus was mostly *centralized*, some were *decentralized* (with some



hierarchy), and only a few were *distributed* (no central or leader node).

One of the most common ways to orchestrate the Cloud is to use (containers) orchestration, especially with Kubernetes, even for the Fog or Edge context [91, 106, 107, 108, 109, 110]. In [111], the authors define three requirements for container orchestration for Edge nodes in the Edge-to-Cloud continuum:

- compatibility with modern container standards, or provision of an equivalent
- securitization of the communications between the layers of the Cloud and the Edge nodes.
- low resource requirements to be run on Edge nodes

These requirements allowed them to create a lightweight container orchestrator called FLEDGE.

In fact, there are a lot of different approaches [112], both in the academic world and in the industry, with some still in use or being developed. Manaouil et al. [113] presents how three approaches to revise Kubernetes original code to better deal with the geo-distribution, i.e., Kubefed [114] and Submariner [115], which do not deliver the illusion of a single Kubernetes system and KubeEdge [116, 117], which is not decentralized, as it is the case for one of the solution for OpenStack, StarlingX [118]. And in any of these cases, the goal was to change the vanilla code of Kubernetes, which we do not want.

We will now present other relevant approaches.

5.1.1 Decentralized and Fault Tolerant Cloud Service Orchestration [119]

This paper from Spataru presents an approach to decentralize the CloudLightning project, a framework to orchestrate Cloud Services with the goal to use it upon Edge resources owned by individuals or even small-scale clusters. Though the Edge is not mentioned in this paper, the project call the resources at the Edge *heterogeneous Cloud* and they mention the Fog Computing, which also corresponds to what we, in this thesis, call Edge Computing.

In CloudLightning, there are two different components, namely *Gateway Service* and *Resource Manager* which use physical resources thanks to an hypervisor. The Gateway Service maintains a catalog of services, described in TOSCA (Topology and Orchestration Specification for Cloud Applications, which has been extended specifically for Fog and

Edge deployments [120]), to compose an application. Users send a description of their requirements and abstract services to form an application and the *Service Optimization Engine* in the Gateway Service choose the best implementation (topology) among the catalog and depending on the availability of the system. Afterwards, the orchestrator reads the topology and deploys it, after which it will monitor the deployment and execution of the involved services, restarting them in case of failure. In the Resource manager, there is a service to “Plug&Play” the hypervisor which could deploy VMs or containers, and the hypervisor is registered in the Self Organizing Self Managing (SOSM) system. There is also a telemetry service which stores data from a telemetry client which has to be connected to this system. Finally, bare metal servers can also be used through configuration.

The SOSM system is self-healing and aggregates monitoring information to assess the performance of the computation resources depending on the business objectives. It keeps a suitability index to guide the requests towards better resources.

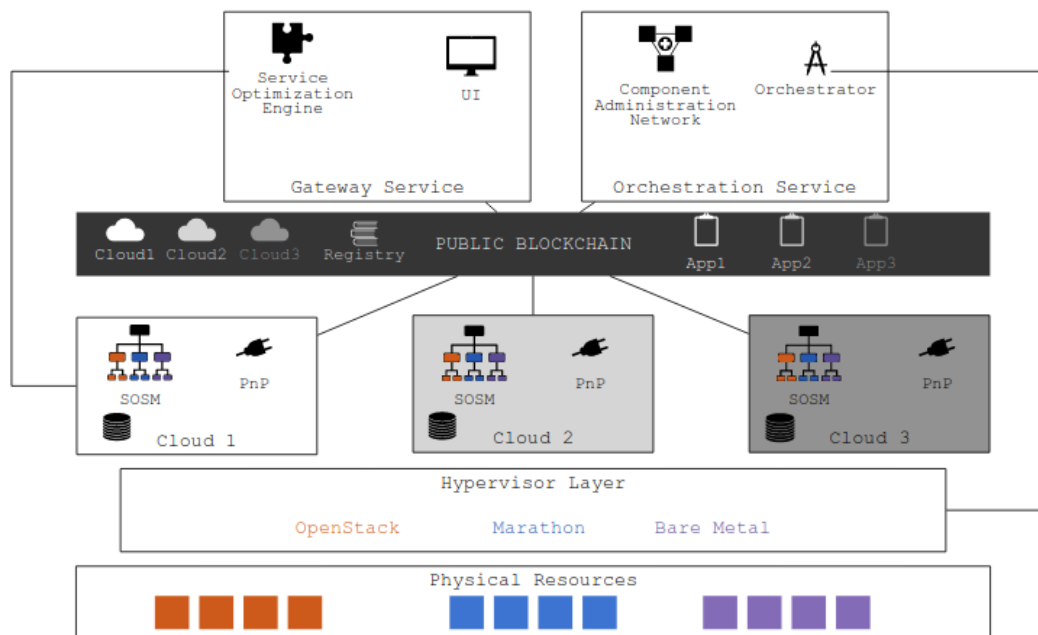


Figure 5.1 – Augmented Decentralized CloudLightning Architecture

When a service fails, as mentioned, the Orchestrator will restart it and the services depending on it will need to have the endpoints updated to be able to keep communications.

There is a capability of decentralizing CloudLightning components, which will be able



to be discovered through [smart contracts](#) on a public blockchain. This augmented architecture is presented in [Figure 5.1](#), with the decentralized components:

- the registry contract keeps information on Cloud Providers, Services and Applications
- the cloud contract keeps resources description (as well as their prices) and endpoints for the scheduler and Plug&Play components
- the application contract tracks deployment status and payments

Finally, the paper presents the Component Administration Networks which make the link between Smart Contracts and Software components, as well as giving monitoring and fault-tolerance capabilities to sets of replicas. This component is composed of three layers. The first one is the underlying [P2P](#) network of nodes that collaborate and maintain the Ethereum blockchain. The second layer manages the nodes thanks to a leader of the network, that controls the transactions related to the component. The third layer administer the components (such as (de)registration of components, assign them some workload, etc.)

Comparison

This approach is non-intrusive, using either VMs, containers or bare metal, so it is also generic.

It does not follow the principle of local-first, as the services will be separated depending on the best placement available, but those services are allowed to collaborate through their configuration.

In case of a network partition, the services on the involved node will probably be restarted elsewhere, but then the application on the involved node will probably not be able to function as it would not have all the services available.

Requests can be redirected to “better” computational resources dynamically, but not on the user decision. The users only define their requirements for the application.

This approach is fully decentralized and uses a blockchain to keep the information on this decentralized approach.

Conclusion

In conclusion, this paper presents an enhance version of the CloudLightning framework, to provide a decentralized orchestrator of Cloud Services in the context of computing

resources composed of home computers or small-scale data centers. The main goal of this paper is to pave the way for a decentralized Cloud with individually owned resources serves as compute nodes, and ways to deal with the payment of these resources.

The originality of this approach resides mainly on the use of a blockchain and smart contracts to decentralize the control on the orchestration.

5.1.2 Liquid computing and Ligo [74]

Ligo is a solution to transparently manage different clusters in Kubernetes, orchestrating them and allowing (computing) resource sharing between those clusters. The authors of the paper envision their approach, *liquid computing*, as a continuum of resources and services to allow transparent and infrastructure-agnostic deployment of applications.

For each workload, users can assign constraints such as geographical locality, cost, capabilities, etc. Their approach derives from a P2P approach, with no centralized control and no intrinsically privileged members. Each actor (owner of a cluster) is the overall infrastructure keeps the full control of their own infrastructure and decides how many resources and services they share and with whom. As in usual P2P system, the topology is fluid and able to manage dynamic changes, with frequent and sometimes unexpected connections and disconnections.

It is built on different key concepts used for the liquid computing. The first one is the discovery and peering functions. The peering in their approach is a unidirectional resource and service consumption relationship, which provides flexibility, but which can be combined to support bidirectional peering. Second, they distinguish nodes as local (attached to the consuming cluster), virtual (abstracted and pooled resources) and big (for nodes with much more capabilities than other average nodes), which allow them to envision different types of clusters and a hierarchical representation of the resources. Third, to achieve robustness, tolerance to network disconnections and scalability, they introduce the concept of resource reflection: objects exist in their *native* form in the local cluster and in their *shadow* form remotely. Finally, the need for network, and storage and data continuum, where, in the first one, the overlapping of different networks across the cluster is a problem and should have a network fabric to handle transparently this problem; and in the second one, the workload should follow as much as possible its involved data.

Ligo itself is the project to enforce the liquid computing approach. It extends -and is built upon- Kubernetes without any change in its standard API.



Comparison

The approach is generic regarding the applications that will run in the pods, but in itself, Ligo only works with Kubernetes solutions, even though their key concepts could be applied to other orchestrator.

Each cluster works independently if need be, so it follows the rules of local-first and collaborative-then, and thus a site can be used during network partitions.

Ligo's approach needs some configurations to establish P2P, but afterwards, the attribution of clusters is dynamic. The location of execution of a request is chosen automatically per scheduling, but *intent-driven*, defined by users.

The goal of the approach is to offload pods on other sites by defining unidirectional peering links, that can be combined to get a real P2P approach.

Conclusion

The liquid computing concept and its associated project, Ligo aim at envisioning a continuum of resources and services in a transparent and infrastructure-agnostic manner. The authors of the paper defined their vision, in which the main characteristics are intent-driven definitions of the workloads, decentralized architecture, multi-ownership and fluid topology.

From these principles, they deduce some pillars, key concepts, to allow for the materialization of the liquid computing. Finally, they present Ligo, their open-source project built on top of Kubernetes to implement their vision through multi-cluster topologies.

Though this approach is really interesting in terms of our requirements, the lack of user-defined requests at fine grain is probably what is missing for the requirements.

5.1.3 HYDRA: Decentralized Location-aware Orchestration of Containerized Applications [76]

HYDRA [76] is a Proof of Concept (PoC) decentralized and location-aware orchestrator that manages microservices in containerized applications, written by Jimenez and Schelen.

In this paper, a focus is made on the location-aware version of the orchestrator, even if it can run location-agnostic. HYDRA builds a P2P overlay network of nodes, where a node is any hardware (virtualized or not) that can execute containerized services, and

thus a node is both orchestrator and (computing) resource. Four roles are available for the nodes:

- The entry role is temporary and corresponds to the role assumed when a node receive a request to deploy, remove or modify applications, where it needs to transfer this request to the appropriate nodes on the orchestrator network that will manage the request. The role is removed when the request has been served.
- There are two controller roles, root and leaf, with the latter used only in location-aware control. The root controls the entire application, while the leaves manage only a set of services of the application.
- The service host role is attributed to nodes who host and monitor services, and send heartbeats to the leader controller of those services for service liveness.

Depending on the number of required services, replicas of these services, the need for scaling-out, and other considerations, the number of service host nodes may change during the life-cycle of the application. These roles are not exclusive and a node can assume different roles for different applications.

Users can deploy an application give its definition through a *recipe*, giving information on what and how to deploy, and how it will be managed. To deploy an application, the node in charge of deploying it launch the search for viable nodes (in terms of resource availability with the regards to the services requirements). To find these nodes, the algorithm follows three properties:

- maximize the probability that the search will query different nodes
- ensure that the load is spread across the HYDRA network evenly
- leverage the information on the distance between nodes

Regarding the experiments, HYDRA is able to scale to at least 20 000 nodes, and support almost 30 000 applications running on 15 000 nodes. The orchestrator nodes are able to work autonomously, and the location awareness has almost no impact on the orchestrator performance.

Comparison

Since it uses applications that run in containers, Hydra is non-intrusive and generic to these existing Cloud applications.

The search for (computing) resources algorithm is randomized or use new nodes as much as possible, so it is difficult to envision it is a local-first approach. Some services



are deployed close to each other, but this is as local as it gets. On the other hand, it is a collaborative-then approach in the sense that all resources can be used by any node. The evaluation proves that the orchestrator itself still function in case of network partitioning.

This approach can be considered dynamic because it requires a configuration of the services, but then the requests are probably routed depending on the application meta-data, since some services are replicated and so forth. Because of that, the location is pre-defined from the users choice for their applications, but afterwards, it is not chosen per-request.

The approach is designed to be fully decentralized, and all have nodes can have the same privileges, even though they have different privileges from the viewpoint of one application.

Conclusion

HYDRA is a decentralized orchestrator for containerized microservices based applications. It allows for location-aware deployment and robust control of these applications as well as management of heterogeneous Edge resources. Finally, it provides resiliency of the deployed applications during their life cycle.

This paper has many interesting approaches to solve problems that the Edge can bring and follows a lot of our requirements. Nonetheless, the location-awareness of the orchestrator does not allow users to finely define the execution location of their request and it lacks the requirement of local-first.

5.2 Control planes and other managements

Some management tools do not fall into the orchestration category. For example, *control planes* are a layer in an application that will manage higher functionalities such as monitoring, scaling, security, configuration, etc. to make sure that the application runs properly and as expected. It is usually represented on top of the *data plane*, which it controls, as this one is the layer in charge of processing data requests, taking orders from the control plane.

In this section, we will explore management solutions that can be seen as these parts of the applications or just do not fit the orchestration category.



5.2.1 OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures [49]

This paper from Saurez et al. presents a *hybrid* control plane for Edge infrastructures. Like us, they consider infrastructures composed of different micro-data centers, maintained by ISPs, with heterogeneous computation resources. They distinguish *coordinated* applications which requires collocation and coordination between different instances, such as collaborative assisted driving and geo-distributed multiplayer games, and *standalone* applications which can work with a single instance, such as virtual reality. This approach is really interesting in the context of the Edge, because it is easier to know where to put applications, depending on these classes. Some of the requirements they observed are also somehow close to ours:

Autonomous control for latency-sensitive standalone applications, with single-site deployments

Coordinated control for coordinated applications, with multi-site deployments

Spatial affinity for applications location sensitivity

End-to-end latency Service Level Objective enforce the guarantees

Dynamic resource allocation for re-deployment in cases of mobility, failures, computing resource scarcity

Most of the differences have to do with the automated placement and handling of users requests, and this is mainly where we have different approaches. In the context of the Edge (and more largely the Cloud), it is an indubitably good idea and practice, but we chose to rely more on the human intelligence and users and give them only information so they can decide how to make their requests.

Comparison

OneEdge runs applications in containers and VMs, thus, without changing their code. It is generic to existing Cloud applications.

To coordinate applications instances, OneEdge has a centralized controller to get a global view, but in order to keep autonomous, rapid decisions without centralized coordination, an authoritative state is kept on each site. The scheduling decisions are also left to the Edge sites. Thus, most of the logic of the application is the same everywhere, but not everything.



An effort has been made to decentralized the control plane as much as possible, but keeping a central controller where decisions are made (about placement and reallocation/migration through monitoring). The collaborations between the sites are available for all types of resources the control plane manages.

Most of the sites are autonomous, but for the centralized part, since it is in the Cloud, fault tolerance is less important, but they ensure it by having a secondary instance of the controller resubmitting the requests (replicated on both instances) that have not be completed as new requests. It is worth noting though, that the launch of coordinated applications are forwarded to the central controller, which is a problem in case of partitions.

Users can request at any moment to launch an application, and these requests are transferred to the closest site using a discovery service. Thus, the control plane decide itself where a request will be executed, but for each request, dynamically.

Conclusion

OneEdge is a hybrid (in the sense of centralization) control plane that allows some of the decision-making to be made autonomously on Edge sites. It presents an interesting approach to discriminate applications between standalone and coordinated ones, which makes sense at the Edge, because the needs are different. To minimize deployment latency, standalone applications are scheduled autonomously, while coordinated applications requires co-location and thus, their deployment and scheduling is managed in a centralized manner. This hybrid approach of what can be done in a centralized manner or not is a really interesting point that could be useful for other approaches as it reduces the number of calls that will go to a centralized [data center](#). Nonetheless, it is not something I desired for my solution as I want a decentralized and P2P approach.

5.2.2 OpenYurt [121]

OpenYurt is a platform extending Kubernetes to allow users to manage large scale Edge computing workloads. OpenYurt architecture is self-described like so:

OpenYurt follows a classic cloud-edge architecture design. It uses a centralized Kubernetes control plane residing in the cloud site to manage multiple edge nodes residing in the edge sites. Each edge node has moderate compute resources available in order to run edge applications plus the required OpenYurt components. The edge nodes in a cluster can span multiple physical regions, which are referred to as Pools in OpenYurt.

Figure 5.2 is the representation of OpenYurt architecture. In this figure, the blue box represents the original Kubernetes components, and the orange box the OpenYurt components. This is a hierarchical approach in which Kubernetes API server is located in the Cloud, as well as OpenYurt Cloud node management capabilities. At the Edge part of the figure, the orange box represents what is inside one node from any node pool on the right part. Nodes at the Edge are composed of different Kubernetes functionalities, such as Kubelet, KubeProxy, etc., and more importantly of the pods themselves as well as OpenYurt Edge functionalities. For example, YurtHub serve as a sidecar to handle requests from Kubernetes components on worker nodes and direct them to the APIServer. The connections from Edge nodes to the Cloud control plane is managed by YurtTunnel (Server/Agent).

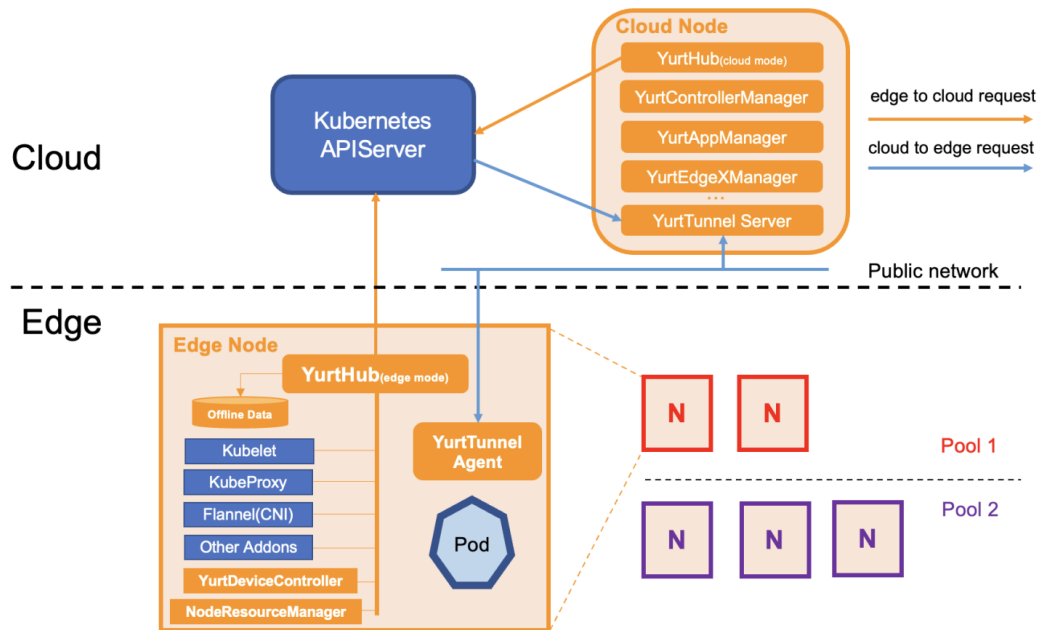


Figure 5.2 – OpenYurt architecture ([Source: OpenYurt's Github](#)).

Comparison

Using the original Kubernetes components, OpenYurt makes non-intrusive enhancements, but the centralized control plane in the Cloud site managing multiple Edge nodes makes it a not really P2P approach. It is generic to any application that can be run on Kubernetes.



Node autonomy is a functionality that allows Edge nodes to continue running even though the Cloud-Edge link is severed, but it is only available through configuration. Moreover, node autonomy only means that users can manipulate existing resources and not create new ones, so its partition tolerance is not complete.

The collaboration are available for every resources managed by Kubernetes.

It is worth mentioning that *service topology* allows users to decide to use in general the endpoints from the same node, or from the same nodepool, so the general location execution of the request (by general location policies) is decided globally by the users, and then the application chose the exact location.

As it as a centralized control plane and a hierarchical architecture, it is not decentralized and is not a P2P system.

Conclusion

OpenYurt is a self-described *platform* that extends Kubernetes to deliver a non-intrusive approach which can allow network partition to some extent (i.e., whatever does not need the control plane). Regarding our requirements, it answers not entirely the local-first and collaborative-then principles, as well as the network partition in node autonomy mode. Moreover, it lacks a decentralized approach and the ability for users to make finely defined requests.

5.2.3 Re-designing Cloud Platforms for Massive Scale using P2P Architecture [122]

In[122] (and [123]), Soares et al. present a way to use OpenStack on a large scale, using P2P architecture. Though it is not designed specifically for the Edge, scaling is the first step towards a solution for the Edge, and the P2P approach checks some of our requirements. They briefly introduce the concepts that were introduced in OpenStack to manage large scale deployments, such as Regions to manage the locality of resources, or Availability zones which allows to logically partition compute, storage or network resources, and finally Cells, which allows to shard several compute resources into cells, and then be dispatched in different locations. But all these solutions had to be *added* into OpenStack code, and thus are available whether you need them or not.

They follow three design principles:

- avoid centralized components as much as possible, for scalability and robustness purposes.
- avoid as much as possible changes in the original code so it is easily adopted and can be more generic.
- externalize problems to an existing outside system as much as possible to simplify the solution and minimize the efforts.

They deploy an entire instance of the Cloud management software on every sites, calling the instance a **cloudlet**.

On each site, they put their agent which can send a request to the local OpenStack or forward it to another site. This agent implements a service proxy on each OpenStack service. Each proxy around a service expose the same **API** as the service they encapsulate, as well as request forwarding and translation logic. This follows the broker based collaboration presented in [Section 3.3](#). Each agent is also composed of an *overlay management* which allow the different agents to discover each other and identify which agent sent a particular request. Each agent has a *state management* to allow them to keep information about users and resources they own. Finally, an interface allows the agents to communicate with each other (Agent-to-Agent communication).

Each agent is responsible to handle every requests that come from any user mapped with it. This mapping is statically created when a user's tenant is created. Through state management, each agent tracks the resources allocated to a tenant, which is required to find resources that are not stored locally but on other sites. They keep they internal ID of the created resources and map them to the cloudlet where the resource has been provisioned. To improve robustness, this state information can be replicated on several agents, but their implementation uses a simple database.

As a **P2P** system, each site maintains a list of their neighbors, where a transitive closure of the graph representing the neighbor relationships returns all sites in the system. More simply put, with every lists of the neighbors of every sites, there is a complete view on the entire system.

Regarding specific services, they use the federated identity for authentication and authorization, which is a service-to-service functionality offered by OpenStack. For the image service, the image is found thanks to the mapping of ID/site presented above. For the scheduling/placement of VMs, they chose to balance the memory load on all sites. Finally, for the network, they do not use a specific approach but mention it is a problem that is being addressed. They mention the Tricircle approach [\[124\]](#), but since this article



is a bit dated, I would also like to mention this approach [7], which was one of the first steps for a part of our own solution.

Comparison

One of their design principles is to avoid touching the code as much as possible, and in the paper, there is actually no mention of having to change the code. In their principles, they also mention that the external approach is to make it generic, but no further fact is given about this, so we consider it to be designed specifically for OpenStack, even though it probably can be adapted for this purpose. In any case, since this approach manages a infrastructure manager, it is generic to any application that could be run in an OpenStack VM.

They operate an entire instance of OpenStack on each cloudlet, so it works perfectly locally on a standalone way. It is collaborative through federation of some of the services and through the agent for the rest.

The P2P approach combined with the fact that there is an instance on each site makes it tolerant to network partition.

The placement of the tenant is determined statically at its creation, but otherwise, other requests are dynamic, following the list of neighbors the agents know. If some of the resource are treated locally as possible; the VMs, which are somehow the principal resource of OpenStack are automatically placed, so we cannot really say it is per user request.

The approach is entirely decentralized and without hierarchical management, designed as a P2P system.

Conclusion

This paper presents an approach to decentralize OpenStack in a non-intrusive and P2P manner, with autonomous instances.

In this approach, images from Glance (the image service) are shared across different sites. They use internal IDs from their agent to keep tracks of these resources shared on different sites with a mapping to IDs created by OpenStack to find the resources locally. This is something really useful for our local-first and collaborative-then principles and has actually been used in our solution.

Since our initial focus was on OpenStack, this paper was obviously important in our early decision making, and especially before we decided we needed a more generic and

user-centric approach.

5.3 Conclusion on managing applications on Edge infrastructure

To ease the reading of the table, we add the following reminder, for orchestration:

[119] Decentralized and Fault Tolerant Cloud Service Orchestration (subsection 5.1.1)

[74] Liquid computing and Ligo (subsection 5.1.2)

[76] HYDRA: Decentralized Location-aware Orchestration of Containerized Applications (subsection 5.1.3)

And for control planes and other managements:

[49] OneEdge (subsection 5.2.1)

[121] OpenYurt (subsection 5.2.2)

[122] Re-designing Cloud Platforms for Massive Scale using P2P Architecture (subsection 5.2.3)

	NI	generic	LF	CT	NP	dynamic	on-demand	decentralized	P2P
[119]	●●●●	●●●●	-	●●●●	●	●●●●	●	●●●●	●●●●
[74]	●●●●	●●	●●●●	●●●●	●●●●	●●	●●	●●●●	●●
[76]	●●●●	●●	●	●●●●	●●●●	●●	●	●●●●	●●●●
[49]	●●●●	●●●●	●●	●●●●	●	●●●●	●	●●	●●
[121]	●●●●	●●	●●●(1)	●●●●	●(1)	●●	●	●	●
[122]	●●●●	●●	●●●●	●●	●●●●	●●	●●	●●●●	●●●●

Table 5.1 – Comparison points on Edge infrastructure management. (1) Only available when activating node autonomy¹.

Table 5.1 represents the comparison of the approaches to orchestrate and manage Edge infrastructure to deploy applications.

The first part (the three first lines) are defined as orchestrators, while the others are respectively identified as only a control plane, a platform and an architecture for an existing management platform.

Whatever the name, they all can be used deploy workflow and applications, via virtualization or containerization.

1. <https://openyurt.io/docs/next/user-manuals/autonomy/node-autonomy>



In terms of existing Cloud applications that are brought to the Edge, OpenStack is used in one of these approaches [122], while Kubernetes is used in two of those approaches ([74, 121] and other aforementioned approaches from the Orchestration introduction. [119] uses an existing project, CloudLightning as a base on which to build upon.

The main requirement lacking in those approaches are the user-centric decisions, but otherwise, some applications make solid candidate regarding our base requirements introduced in Section 3.1, such as [74, 121], though [121] only offers tolerance to disconnections as a *degraded* mode.

Since none of the approaches corresponds entirely to my requirements, we will now take interest in a way that could help to manage the different instances of the application we want at the Edge, and especially the collaborations between them.

5.4 A service mesh at the Edge?

William Morgan from Buoyant Inc. (Buoyant created Linkerd, one of the most used service mesh) defined services meshes as [28](© 2011 IEEE):

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

In practice, a service mesh helps with the separation of the tasks of development from the operations. The developers can focus on the business logic and code, and operators on deploying the application, keeping it running, etc., as put by William Morgan: [the service mesh is] “deployed alongside application code, without the application needing to be aware”. Actually, RedHat specified that [125]:

Without a service mesh, each microservice needs to be coded with logic to govern service-to-service communication, which means developers are less focused on business goals.

In this sense, a service mesh could help with the service collaborations discussed in Section 3.2.

It usually consists in a scalable set of proxies deployed with the application [126], as specified by William Morgan in the quote above. These proxies, composing the *data*

plane, manage the communications between the micro-services, passing them through the *control plane* to execute whichever functionalities the service mesh provides.

Figure 5.3 presents an abstracted service mesh architecture. The control plane and the data plane are represented separated. Each service is deployed alongside a proxy as a sidecar, but the proxy could also be represented as the capsule around the service, because it intercepts incoming and outgoing communications to transfer them to the control plane. These proxies thus serve both as proxies and as reverse proxies [127]. In the control plane, there are different components achieving the functionalities of the service mesh, such as monitoring, service discovery (to build a registry of services), etc. Also, since it is an abstract representation, there can be any number of services or components, which is shown as the dots on the right part of the figure.

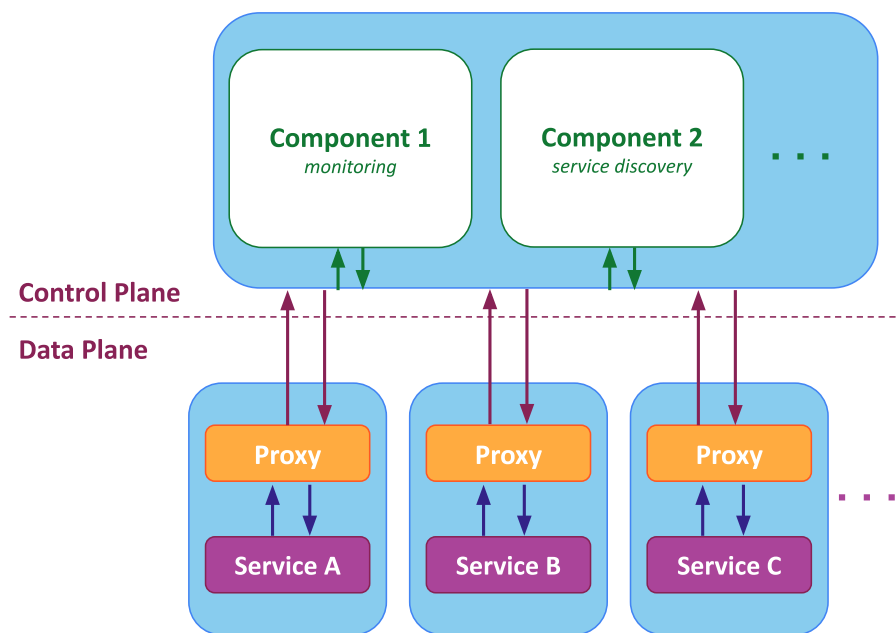


Figure 5.3 – Architecture of a typical service mesh, inspired from [30]

Though service meshes were not created in the first place for the Edge, they are a very important software architectural framework for Edge computing, because they could bring functionalities such as service discovery, load-balancing, resiliency, scalability, low-latency offloads and privacy [128, 28].



5.4.1 Istio [129]

Istio is an open source service mesh that uses Envoy² as the proxy. Envoy is a proxy often used as a dataplane³ and is deployed as a sidecar along each service. Envoy has a L3, L4 and L7 filter layer, which allows to filter and route TCP/UDP, TLS (see [Transmission Control Protocol](#), [User Datagram Protocol](#), [Transport Layer Security](#) for definitions), HTTP(/2) and gRPC requests, with much more functionality on the last two.

Istio provides traffic management through Envoy and service-level properties (time-outs, retries, etc.) configurations, through the Istio Pilot agent running in the sidecar or the gateway container. The control plane also provides observability through the tracking of requests, which allows for example to track dependencies between services, how they interact between each other, and also metrics to control the system. Historically, the control plane was composed of four components, namely Pilot (service discovery), Galley (configuration), Citadel (certificate generation), and Mixer (extensibility). But since in 2020, Istio has been delivered as a signal binary, Istiod, to simplify the process of deployment and because a lot of the benefits of micro-services did not make sense in Istio context.

Figure 5.4 represents a multi-cluster version of Istio: the mesh covers three different clusters (in orange) on two different networks (in black). Traffic goes through Envoy (the pink hexagon), whether it is inside a cluster (green arrows), or between clusters (dotted blue arrows). If the West-North cluster goes down, the West-South can take the load temporary. Those two cluster communicate with each other thanks to being in the same network. In such a configuration, all services are shared, and if they share the same namespace, they will be considered as a single combined service.

Comparison

It is mostly generic to applications that can be used in Kubernetes⁴. Since it extends Kubernetes, there is no changing the code for many applications, however, some might require some changes, depending if the application requirements are met or not yet. This requirements *for some applications* have given Istio a three/two clouds for non-intrusion, for fairness.

Since multi-cluster is only supported with a central control plane, we do not consider that it can survive a network partition. Of course, though, if one site is cut off from

2. <https://www.envoyproxy.io/> - Accessed: 2022-09-20

3. <https://servicemesh.es/> - Accessed: 2022-09-20

4. <https://istio.io/latest/docs/ops/deployment/requirements/> - Accessed: 2022-09-20

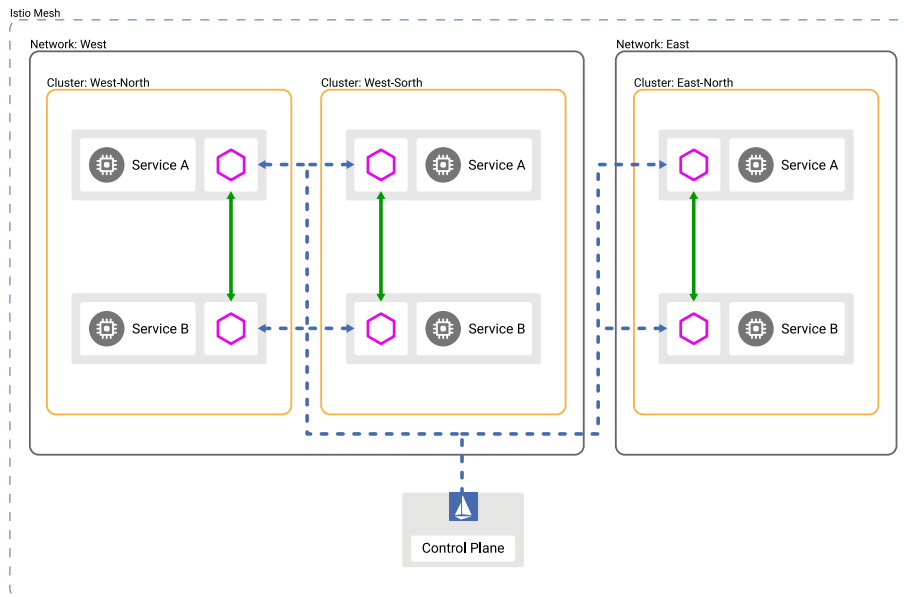


Figure 5.4 – Multiple clusters on Istio

the rest, users can still use another site. In mesh federation⁵, it might be possible, but even with the Google demo⁶, the information given on the functionality is not sufficient to determine this (especially because it seems to mostly allow service spreading across potentially different sites, but at minimum different meshes).

The configuration of the deployment is prepared statically, and can be updated depending on the users needs.

The location of execution of requests is determined through Envoy load-balancing, so it is automatically chosen.

Since Istio has been designed for the Cloud, its control plane is centralized, and thus, most of the functionalities are executed there. Finally, there is no real point in talking about Istio P2P capabilities, except for this example⁷, where they achieved a multi-mesh, multi-cluster which can be considered as a P2P version of Istio, so it is technically possible, though not designed for this.

5. <https://istio.io/latest/docs/ops/deployment/deployment-models/#multiple-meshes> - Accessed: 2022-09-20

6. <https://github.com/GoogleCloudPlatform/istio-samples/tree/master/multicluster-gke/dual-control-plane> - Accessed: 2022-09-20

7. <https://banzaicloud.com/blog/istio-multi-mesh/> - Accessed: 2022-09-20



Conclusion

Istio is a service-mesh using Envoy as a sidecar proxy for every services. Some efforts have already been made to allow multi-cluster in Istio, though it is still not designed to be entirely autonomous and decentralized as the control plane is centralized (at best one per region). Thus, it is going in a really good direction for us, with some possibilities to make a P2P version, it is still lacking user chosen locations for executing requests, as it is done automatically.

5.4.2 Linkerd [130]

Linkerd is another service mesh, which was the first called like that.

Linkerd, like Istio, allows to filter and route [TCP](#), [TLS](#), [HTTP\(/2\)](#) and [gRPC](#) requests, with much more functionality on the last two.

Linkerd2 uses Linkerd2-Proxy⁸, an “ultralight micro-proxy” tailored for their needs to make Linkerd smaller and simpler. Linkerd developers decided that Envoy had too much functionalities which made it inappropriate for a sidecar use. Whether it is for the complexity, the resource consumption, or the danger of having such a complex proxy introducing security issues in its code, they preferred a lightweight approach that work perfectly for their use case. The downside is that it is not recommended to use Envoy as a proxy and Linkerd2-proxy would not be usable by another service mesh⁹.

[Figure 5.5](#) shows an overview on how Linkerd deal with the Multiple Cluster functionality. Clusters (here, west and east) communicates with each other through a Multi-Cluster gateway. This gateway is then responsible to route a request from a Service from the other cluster to the involved Service in its own cluster.

Comparison

Linkerd, as for most service meshes, does not require any change in the business code of the underlying applications, though it is also specific to Kubernetes, because it only consider resources managed by Kubernetes (with mesh expansion on the roadmap¹⁰).

It is design to work locally first since it is supposed to be running in the Cloud, but in multi-cluster, it is possible to get resources from other services.

8. <https://github.com/linkerd/linkerd2-proxy> - Accessed: 2022-09-20

9. <https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/> - Accessed: 2022-09-21

10. <https://github.com/linkerd/linkerd2/blob/main/ROADMAP.md> - Accessed: 2022-09-21

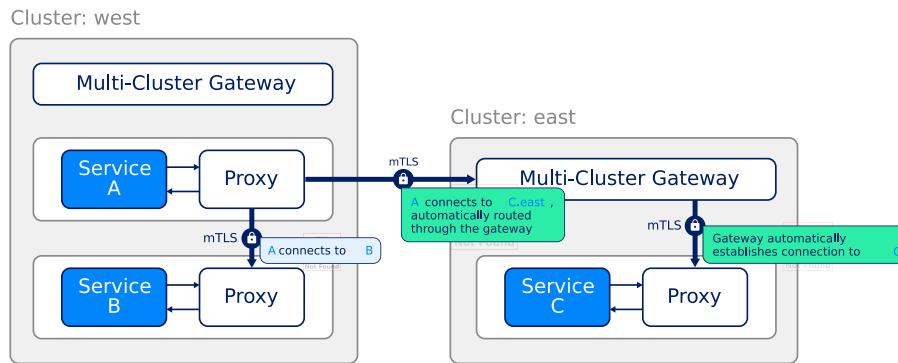


Figure 5.5 – Multiple clusters on Linkerd

The recommendations on how to make an architecture for multi-cluster Kubernetes give these architectures with Linkerd a solid chance of surviving network partitions¹¹. They recommend to maintain independent states between clusters and manage updates via replications so updates are only where required. They also advise to keep the control planes independent separate, which further makes the system more tolerant to network partitions. When the control plane is down, though, the proxies will use the information they already have and fall back to **Domain Name System (DNS)** if a request is supposed to be routed to a service they do not know. But this implies that any proxy deployed when the control plane is down will need to timeout all new requests until the control plane is up again¹². In conclusion, Linkerd in multi-cluster mode is tolerant to network partitions, but not entirely.

Regarding dynamicity, Linkerd uses an algorithm called Exponentially Weighted Moving Average (EWMA) to automatically route requests to the fastest endpoints¹³. The execution location of a request is chosen for the user, in part through traffic split.

As for Istio, we cannot consider this application of the **P2P** point, since it has not really been designed to work with several instances of the application.

Conclusion

Linkerd is a service mesh that relies on a homemade, lightweight and less generic proxy than Envoy, called Linkerd2-Proxy. It also supports a multi-cluster architecture if need

11. <https://linkerd.io/2020/02/17/architecting-for-multicluster-kubernetes/> - Accessed: 2022-09-21

12. <https://linkerd.io/faq/#what-happens-to-linkerd-s-proxies-if-the-control-plane-is-down> - Accessed: 2022-09-21

13. <https://linkerd.io/2.12/features/load-balancing/> - Accessed: 2022-09-21



be, through gateways on each cluster that redirects the request they receive locally to the required service.

5.4.3 Conclusion on service meshes

We studied two different service meshes which mostly have the same functionalities and goals, except that Linkerd seems to be more focused on being lightweight. In the appendix (Chapter 12), we will also talk a bit about Consul, which has been considered as a solution for us for a time.

















	NI	generic	LF	CT	NP	dynamic	on-demand	decentralized	P2P
Istio									-
Linkerd									-

Table 5.2 – Comparison points on service meshes.

Overall, service meshes could be a good solution to our main research question of putting Cloud applications to the Edge if they were designed for the Edge, to be more collaborative, more decentralized, and more generic. It is especially true since they are doing more and more efforts to support multi-clustering (both propose versions of this functionality, with different levels of support) and more application managers.

It is also noteworthy that despite some efforts, service meshes are pretty demanding and impact the performance of small devices, so there is a lack in service meshes that would be able to be used at the Edge [128]. A lightweight version that checks our requirements might be the solution to our research questions, and especially this one: “since service meshes are designed to manage the communications between services of an application outside of its business code, can it be a solution to using Cloud applications on Edge infrastructures without changing their code?”.

As we did not find an approach that fits entirely the requirements, we will now consider techniques to make applications at the Edge in order to learn if some ideas could help us in building a generic solution on the main question raised in this manuscript.

HOW TO MAKE EDGE APPLICATIONS NATIVELY



Though we think it is better to not reinvent the wheel, and so use as much as possible Cloud applications directly on the Edge, it is still possible to develop an Edge application from scratch. First, it makes sense of course simply for new applications that will be developed with the Edge in mind. Second, some frameworks might have been developed to do so, in case some of them might automatize the process of pushing an application to the Edge, which could work on an existing Cloud application. Third, some approaches might give guidelines on how to properly develop an application for the Edge [131], which is of interest even to know how to push an existing Cloud application to the Edge.

6.1 Towards Scalable Edge-Native Applications [51]

[51] from Wang et al. presents an approach to “edgify” an application, i.e., the Gabriel platform [132], that was designed for a single user. Though this approach is mainly about multi-tenancy, because Gabriel was already an application to offload the functionalities of IoT to Fog/Edge, it gives requirements for the Edge.

Since the authors are changing Gabriel to be more scalable, they talk about the enhance version as "Scalable Gabriel", while the other is "Original Gabriel". In this paper, they refer to a three tier architecture of the Cloud computing. The Tier-3 would be the Edge, composed of IoT and mobile devices, and where data are pre-processed in the Gabriel front-end (e.g., compression and encoding). The Tier-2 would correspond to the Fog (what we refer to as the Edge), composed of cloudlets, small DCs, etc., and where the Gabriel back-end and its different modules treat the data from the Tier-3. The Tier-1 would be the Cloud itself, with further and bigger DCs, with more compute power, energy usage and elasticity.

The original Gabriel was built for a single user, i.e., only one sensor interacting with



one cloudlet (1:1), and they want to allow several devices to interact with the same cloudlet (n:1).

Several applications run on the original Gabriel Platform, but the authors focus on Wearable Cognitive Assistance (WCA, running typically on glasses for Augmented Reality in their cases) applications because they deem that these applications present three crucial characteristics for Edge computing:

- they transmit large volumes of (video) data from the device to the cloudlet (Tier-3 to Tier-2, or in our case, we would say from the devices to the Edge).
- they have strict latency requirements
- they impose high computing demands on the cloudlet, especially in the GPUs.

Regarding the last characteristic, the workflow of these applications in the cloudlet is composed of two phases. The first phase treat the images to extract an abstract, symbolic representation of the state of the task, while the second phase, far less compute intensive, operates on this symbolic representation, implementing the logic of the task.

To tackle the scalability problems, they first reduce the load going through the wireless network and to the cloudlet through adaptation (*Workload reduction*); the second is about a better scheduling of cloudlet resources to minimize queuing and impacts of potential overloads (*cloudlet resource allocation*). Both those approaches are tested afterwards.

To adapt the original Gabriel to these points, they leverage the WCA applications characteristics. The first is the *human-centric timing*, which corresponds to the fact that the applications have active phases, where they need to sample and process video frames to give instructions to the users, which will perform these instructions in the passive phase of the applications. In this passive phase, the sampling and processing of video frames to determine if the user has soon finished the instructions, which would considerably reduce the load.

The second is *event-centric redundancy*, where they detect redundant frames that do not provide new information to reduce the use of the wireless bandwidth and the cloudlet cycles on frames.

The third is *inherent multi-fidelity*, in which they leverage the fact that some of the processing algorithms can trade-off fidelity and computation. When a cloudlet becomes overloaded by multiple applications, it is possible to act on this trade-off to ensure the applications functionalities.

Finally, they give a taxonomy of relevant adaptations depending on the characteristics of the applications. These adaptation techniques are presented in [Table 6.1](#).



	Question	Example	Load-reduction Technique
1	How often are instructions given, compared to task duration?	Instructions for each step in IKEA lamp assembly are rare compared to the total task time, e.g., 6 instructions over a 10 minute task.	Enable adaptive sampling based on active and passive phases.
2	Is intermittent processing of input frames sufficient for giving instructions?	Recognizing a face in any one frame is sufficient for whispering the person's name.	Select and process key frames.
3	Will a user wait for system responses before proceeding?	A first-time user of a medical device will pause until an instruction is received.	Select and process key frames.
4	Does the user have a pre-defined workspace in the scene?	Lego pieces are assembled on the Lego board. Information outside the board can be safely ignored.	Focus processing attention on the region of interest.
5	Does the vision processing involve identifying and locating objects?	Identifying a toy lettuce for a toy sandwich.	Use tracking as cheap approximation for detection.
6	Are the vision processing algorithms insensitive to image resolution?	Many image classification DNNs limit resolutions to the size of their input layers.	Downscale sampled frames on device before transmission.
7	Can the vision processing algorithm trade off accuracy and computation?	In image classification, MobileNet is computationally cheaper than ResNet, but less accurate.	Change computation fidelity based on resource utilization.
8	Can IMUs be used to identify the start and end of user activities?	User's head movements are of significantly higher magnitude when searching for a Lego block.	Enable IMU-based frame suppression.
9	Is the Tier-3 device powerful enough to run parts of the processing pipeline?	A Jetson TX2 can run MobileNet-based image recognition in real-time.	Partition the vision pipeline between Tier-3 and Tier-2.

Table 6.1 – Application characteristics and corresponding applicable techniques to reduce load (Source: Wang et al. [51]).

Then, they proceed in testing these techniques for workload reduction, as mentioned above. From these evaluations, the authors are able to conclude that Edge-native applications should be written in very different way from Cloud-native applications to be scalable.

Comparison

Since this paper is about specific use-cases (WCA applications) that requires some computing locally and most of it in what we call Edge computing, the question of collaboration is difficult, since collaborations is only vertical (from one Tier to another) and not horizontal (between multiple sites of the same Tier).

The approach is about modifying an existing application (original Gabriel), but the overall logic did not change. It is generic for any (WCA) application that can run on the Gabriel platform only, but the adaptations are much wider and could be used both to adapt a Cloud application or to build and Edge application.



Monitoring of the resources is done on both tiers of the involved architecture (Tier-3 and Tier-2), but some are monitored at Tier-3, others at Tier-2. As said before, the collaboration is only vertical and most resources are not managed locally, so it is only some part of the computation that is available locally **and** it only a subset of resources is used in collaboration.

Once again, because of the verticality, network partition is tricky. If we consider network partition in the same tier, it will not be a problem, since there is no collaboration inside. But, in the context of WCA applications, on a wireless network, there is a even larger chance of a partition between any of the devices and the cloudlet, and thus, any offload computation could not be done. Thus, only one cloud is awarded, for the case of partition in the same tier.

The authors do not debate the n:n relationship between devices and cloudlets, so there is no real dynamic choice of the location of execution of requests, and the same can be said for the user ability of choosing the location.

The cloudlet is the actual only place of execution of the tasks, no mention of a re-balancing to other cloudlets, so the approach is hierarchical and centralized.

Because the Tier-3 Gabriel front-end is not the same as its back-end Tier-2 counterpart, and most of the workload is done in the cloudlet, we cannot say this approach is a P2P system.

Conclusion

In conclusion, this approach is a redesigned version of Gabriel, a platform to allow a single wearable device process data in a cloudlet, to allow the cloudlet to manage several devices. By degrading the quality of service when it is possible and not required, the cloudlet is able to withstand a much higher load and number of connected devices than previously, as well as the wireless network load is decreased. They defined a taxonomy of applications and a set of rules to apply to allow a smarter processing of the data recorded by edge devices, which could be reusable to adapt a Cloud application for the Edge as well as to develop a new Edge-native application from scratch. This approach, by only considering only vertical relations (edge-devices to cloudlet), is not using the entire capabilities of the Edge and is designed specifically for small edge devices that need to offload almost all their workloads to the Edge servers/Fog.

This solution is pretty interesting for the ideas on how to adapt an application for the Edge, even though it is mainly aimed at WCA applications because they leverage some

of their unique characteristics. One conclusion in this paper I found really interesting is they consider that Edge-native applications should be written very differently from Cloud-native ones to be scalable. It is probably true for applications running on the Edge devices, but it would be interesting to test on Edge servers, since it would be that we cannot have a non-intrusive approach.

6.2 Highly-Available and Consistent Group Collaboration at the Edge with Colony [50]

Colony [50, 133] by Toumlilt et al. is composed of a middleware and a database to address the lack of approaches for collaborative applications at the Edge.

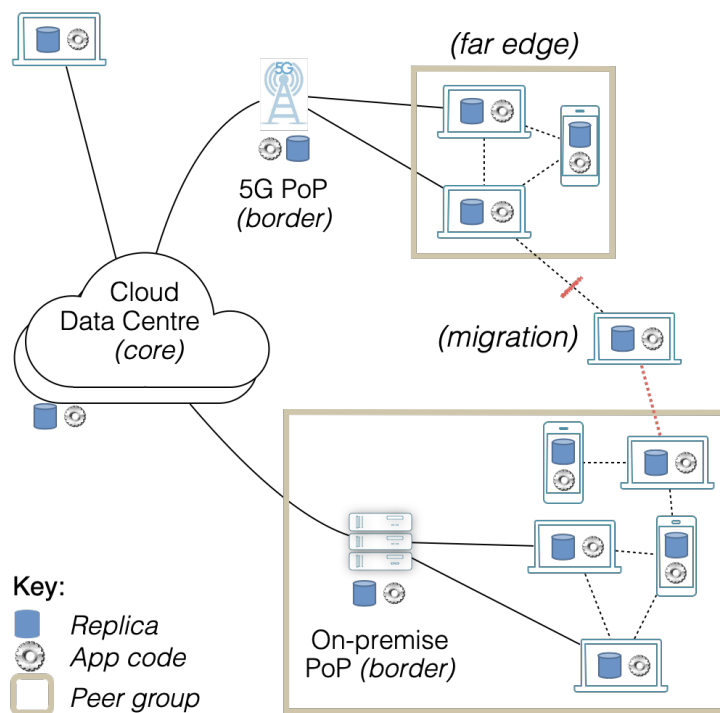


Figure 6.1 – An example of a Colony topology (Source: [133]).

The database responds to the need of Edge-to-Cloud continuum and is designed for collaborative applications. In a typical usage, presented in Figure 6.1, the Cloud core is composed of a small number of DCs, and the Edge nodes are grouped by proximity, connecting either to the core DCs or to a PoP server at the border of the group. Due to mobility, a node can migrate from one group to another.



The focus of this work is consistency in the Edge with a local-first approach. As such, they enforce a hybrid approach for consistency: globally, the model follows a Transactional Causal Plus Consistency (TCC+) introduced in the paper; in the data centers, the consistency provided is Snapshot Isolation (SI), as the servers are connected through high-quality networks; finally, in the local groups, they also enforce SI, but the system needs to be able to work disconnected to keep them available and without losing the overall TCC+ guarantees, using EPaxos.

The TCC+ model guarantees causal consistency [134], rollback-freedom (no rollbacks after a value has been read), strong convergence (any two nodes observing the same set of updates read the same value), atomicity, snapshot. It extends the TCC model [135] with the strong convergence and rollback-freedom

Comparison

The Colony middleware is designed to provide a simple API to develop and deploy collaborative applications at the Edge. This means that some effort is made to help developers to make their applications collaborative at the Edge, but it is thus a very intrusive method to follow the necessary data types and API. This approach is generic to any type of collaborative applications for the Edge.

The local-first principle is one of the focus of the approach, and it is enforced by replications and caches. There are collaborations (in our sense) between the nodes in the same group especially when needed, on any types of resources.

The application is supposed to be able to work offline with the entire system and design, so in effect, it supports network partitions: “the application can read and write data, even when disconnected from any remote server”.

The required information are transmitted to required nodes dynamically, even in case of mobility. The location of execution of requests is mainly local, and on the peers of the collaborative applications. Data is replicated and cached depending on the edge client *interest*.

The approach is hierarchical, as some transactions are run only in the core Cloud, such as analytics or large queries. The core also manages the authentication of nodes via the session manager, but it is only for the current implementation.



Conclusion

In conclusion, Colony is an approach to design collaborative applications for the Edge, guaranteeing the highest consistencies possible for the availability of the resources in the context of the Edge. To do so, it follows a hybrid approach in consistency, using TCC+ in the global cases and Snapshot Isolation in the peer, Edge groups. It allows migration of nodes between groups for mobility and ensures fault-tolerance and scalability by having the groups replicated roots in the Cloud.

Overall, it is a really interesting approach to design collaborative applications for the Edge, which is robust, mostly decentralized, generic and follows the base principles of local-first and collaborative-then.

As a final note, this approach uses a shared database approach, which we did not want to use for reasons presented in [subsection 3.3.1](#), but makes sense if properly used for a new application, developed from scratch.

6.3 Conclusion on ways to make an application at the Edge

To ease the reading of the table, we add the following reminder:

[\[51\]](#) Towards Scalable Edge-Native Applications ([Section 6.1](#))

[\[50\]](#) Highly-Available and Consistent Group Collaboration at the Edge with Colony ([Section 6.2](#))

















	NI	generic	LF	CT	NP	dynamic	on-demand	decentralized	P2P
[51]						-	-		
[50]									

Table 6.2 – Comparison points on making applications at the Edge.

[Table 6.2](#) sums up the comparison on the approaches to make applications at the Edge.

Both approaches are pretty different and do not focus on the same requirements.

In [\[51\]](#), the focus is about making intelligent choices (both in design and by the use of adaptive algorithms) to lower the load when possible and allow several applications to run on the same node. Thus, it also gives clues on how to adapt an existing Cloud application for the Edge, but keeping only its functionality logic and changing entirely the way it is used.



In [50], the authors focus on bringing consistency, scalability and offline mode to collaborative applications at the Edge. Moreover, the approach follows the local-first principle, same as us, which makes it a particularly strong candidate to answer our requirements, but it lacks the decentralized, P2P system that leaves the users the ability to choose the execution location of their requests.

Both answer parts of the puzzle of building an Edge-native applications.

COMPARISON AND CONCLUSION ON THE STATE OF THE ART



7.1 Comparison

To ease the reading of the table, we add the following reminder, for approaches that can't be in a single word:

- [119] Decentralized and Fault Tolerant Cloud Service Orchestration (subsection 5.1.1 - Decentralized CloudLightning)
- [122] Re-designing Cloud Platforms for Massive Scale Using a P2P Architecture (subsection 5.2.3 - P2P OpenStack)
- [51] Towards Scalable Edge-Native Applications (Section 6.1 - Scalable Gabriel)

Approach	NI	generic	LF	CT	NP	dynamic	on-demand	decentralized	P2P
[119]	●●●●	●●●●	-	●●●●	●	●●●●	●	●●●●	●●●●
Liqo [74]	●●●●	●●	●●●●	●●●●	●●●●	●●	●●	●●●●	●●
HYDRA [76]	●●●●	●●	●	●●●●	●●●●	●●	●	●●●●	●●●●
OneEdge [49]	●●●●	●●●●	●●	●●●●	●	●●●●	●	●●	●●
OpenYurt [121]	●●●●	●●	●●	●●●●	●	●●	●	●	●
[122]	●●●●	●●	●●●●	●●	●●●●	●●	●●	●●●●	●●●●
Istio [129]	●●●●	●●	●●●●	●●	●	●●	●	●	-
Linkerd [130]	●●●●	●●	●●●●	●●	●●	●●	●	●●	-
[51]	●	●●	●	●●	●	-	-	●	●
Colony [50]	●	●●●●	●●●●	●●●●	●●●●	●●●●	●●	●●	●●

Table 7.1 – Comparison points on the overall state of the art.

Table 7.1 represents the overall comparison according to the different categories.

There is no solution that fits entirely the requirements, but there are some that fit perfectly or almost our principles of local-first and collaborative-then [50, 74, 49, 121, 122].



A lot of the different solutions considered overall had a non-generic approach, except for how to build a Edge-native application, of course.

A requirement that is also mostly overlooked is the resilience to network partitions, with four approaches that do not have a way to cope with it, even when we do not count service meshes that were designed for the Cloud (and thus need it less). In the context of the Edge, it is unreasonable to neglect site autonomy when disconnections are the norm rather than the exception.

In terms of putting Cloud applications to the Edge, the solutions studied leverage a distributed database in two cases [119, 50], while others use some sort of brokering or service-to-service [122, 51] without the location-aware aspects to make it less intrusive. And interestingly, the non-intrusive requirement is the one that is mostly observed, except for solutions to build applications for the Edge, which makes sense for Edge-native applications.

As for the genericity, it is mainly provided by allowing to deploy containerized applications [74, 76, 121, 130, 129] or applications in containers, VMs, and sometimes more (like bare metal) [119, 49].

As mentioned in the more specific comparisons, the user-centric to have on-demand, dynamically managed requests is what is mostly lacking from the approaches, though dynamicity is considered at request level in three solutions. This is the point that definitely needs to be taken into account for our solution, if we want to allow users to define where they want their requests to be executed.

7.2 Conclusion

In this part, we have seen different solutions to allow applications running at the Edge.

One of the goals was to find original (and different from one another, except for the service meshes), generic and/or usable approaches that have not been studied already by members of our project, such as in Manaouil et al.[113].

This State of the Art does not cover specific subjects in the deployment and life cycle of applications services, such as algorithms in charge of discovery, monitoring, load-balancing or placement, but rather tried to focus on solutions that fit at least some of them, to have ideas on how it is dealt with. Of course, still in the context of having applications at the Edge. It is also sidelining approaches to manage the physical, hardware Edge infrastructure that study how to put computing devices in the infrastructure because it was out of scope,



and we considered that we had the infrastructure ready to use.

Some of the solutions presented here are really good answers to the Edge challenges in their own way, and only fail our own requirements.

The hierarchical approaches, for example, create Single Point of Failure (SPoF) and bottlenecks, and thus cannot be envisioned as solutions unless they have mechanisms to avoid those problems (replication [50], temporary autonomy [129], etc.). The lack of tolerance to network partitions/disconnections is also a problem in the context of the Edge, as mentioned before. Finally, the lack of dynamic and on-demand, only when required collaborations, with most of the workload executed as locally as possible, is what drove the approach we built and will now detail.

What can be taken away from this State of the Art, regarding the requirements:

- **While a lot of approaches are collaborative, not enough respect the local-first principle or the tolerance to network partitions, which is a problem in the context of the Edge.**
- **Approaches that are generic concentrates mostly on deploying containerized applications, especially with Kubernetes, without considering the collaborations that could be done between the deployed application.**
- **Solutions often consider dynamicity of the requests, which is a good thing in a highly dynamic context, but they do not consider enough the will of the users by allowing the execution location to be chosen on-demand by the users.**

None of the solutions answered to all the requirements, but the service mesh approaches could be leveraged to make a service dedicated to managing the geo-distribution in a decentralized and P2P manner, outside of the business code, similar to the multi-cluster approach of Linkerd. This point of view will be more detailed and explained in the next part, with our own solution.

PART III

Developing a solution to use native Cloud Applications at the Edge





For within each seed, there is a
promise of a flower.

*Dillon, Alien*³

In this part, we are going to see how we designed a solution that fulfill the requirements defined in [Section 3.1](#).

In [Chapter 8](#), I present the overview of the approach, and the concepts to give some insights on to reproduce an implementation. This includes the basics on how the approach function, on what it relies from Cloud applications and what is required achieve the requirements we explained earlier. Specifically, I will detail the Domain Specific Language ([DSL](#)) made to allow for outside management of geo-distribution concerns, and an overview of some collaborations it offers and an insight in other possible useful collaborations is also presented. Furthermore, a glimpse into the classification of resources dependencies is given to understand how we can help users with the manipulation of resources.

Then, in [Chapter 9](#), I will expose the vision and current implementation, called Cheops, a service mesh to geo-distribute Cloud applications at the Edge. Finally, I will examine the validation we made on our implementation, that is a common work between Geo Johns Antony, a PhD student colleague, and me.

This part includes work from different communications made in the scientific community [[136](#), [96](#), [31](#), [32](#), [137](#), [138](#)] and the OpenInfra Summit 2022 [[33](#)], along with different people in the team, namely Geo Johns Antony, Ronan-Alexandre Cherrueau, Adrien Lebre, and Javier Rojas Balderrama, without forgetting the implication of Matthieu Simonin.

AN APPROACH DEDICATED TO GEO-DISTRIBUTION



This chapter exposes the logic of the approach, how it works, what is needed to achieve it. First, I will explain how the modularity of applications is one of the main characteristics of this approach by allowing the externalization needed. Then, I will present scope-lang, the Domain Specific Language made to extend requests with the geo-distribution information. Finally, I will explicit what types of collaboration we have envisioned for the solution.

It is worth mentioning that most of this chapter comes from our Euro-Par article [31].

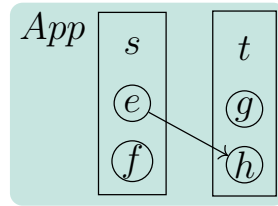
8.1 A service-mesh approach

As we discussed in [Section 1.2](#) and [Chapter 3](#), service-based applications, like a lot of Cloud applications, are composed of different services. These services aim to provide modularity to the applications, and the decoupling of functionalities. This modularity principle comes from software engineering, allowing separation of concerns (for example to have different teams working on separate aspects in the business code), reusing of existing modules, and overall better development manageability.

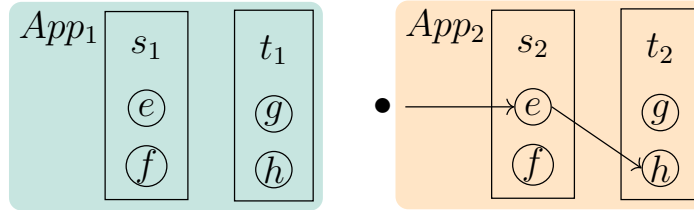
The services communicate between each other through the [API](#) they expose that allows access specific function to manipulate the resources they handle. In particular, a lot of these services use RESTful APIs to make requests to each other, which a resource-oriented paradigm [139, 140, 141].

These are the kind of applications that allow our approach: (micro)service-based Cloud applications with RESTful APIs. These applications expose endpoints to follow the REST architectural styles, allowing users and other services to interact with the services functionalities. The series of calls in an application that execute a particular request is called a workflow.

We discussed previously how the premise of a solution is to deploy an entire application, which automatically checks the local-first principle. Deploying all services of one application constitutes an *application instance*, which we will often use as simply *instance* afterwards. The services running in an application instance are called *service instances*. Each application instance achieves the application intent by exposing all of its workflows to enable the manipulation of resource values.



(a) Application *App* made of two services *s* and *t* and four endpoints *e, f, g, h*. The $s.e \rightarrow t.h$ represents an example of a workflow.



(b) Two independent instances *App₁* and *App₂* of the *App* application. The \bullet represents a client that executes the $s.e \rightarrow t.h$ workflow in *App₂*.

Figure 8.1 – Microservices architecture of a Cloud application

Figure 8.1 illustrates workflows in service-based applications. On the top, Figure 8.1a represents a typical application *App* made of two services *s, t* exposing endpoints *e, f, g, h*. It shows an example of a really simple workflow $s.e \rightarrow t.h$, where a function in service *s* calls an endpoint *h* on service *t*. This is a figure similar to the Figure 1.2 (page 13), but also exposing the endpoints.

On the bottom, Figure 8.1b represents two application instances of *App* and their corresponding service instances: *s₁* and *t₁* for *App₁*; *s₂* and *t₂* for *App₂*. A client (\bullet) triggers the execution of the workflow $s.e \rightarrow t.h$ on *App₂*. The request is addressed to the endpoint *e* of *s₂* which, in turn, handles it and contacts the endpoint *h* of *t₂*. It is important to understand that it is the same exact workflow in both figures; the second figure only have two instances of the same application, and a client triggering the workflow $s_2.e \rightarrow t_2.h$ on the second instance.

As discussed several times, running one instance of a microservices application honors the local-first principle *automatically*. Since Figure 8.1b shows two independent instances, they could be deployed on two different sites, which would be a first step towards a global infrastructure. This is the configuration presented in Figure 8.2, with a request executed on $Site_2$, without impacting $Site_1$ at all.

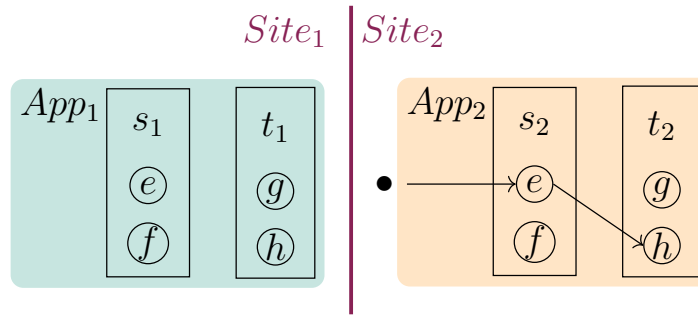


Figure 8.2 – Two instances of a Cloud application on two different sites

Having entire independent instances on different sites means every local requests can be executed on each site, even in case of network partition. However for the global system, it results in plenty of concurrent values of the same resource distributed but isolated among all instances (e.g., two sites manage the same kind of resources but their values differ as time passes). With this configuration, manipulating any concurrent value on any instance requires to code the collaboration in the application and let clients specify how to use it. Moreover, collaborations are required for a single coherent system, as we discussed in Chapter 3.

Furthermore, as mentioned several times, the modular decomposition of the code is popular for programmers of microservices architecture because it divides the functionality of the application into *independent and interchangeable* services [142]. The most important benefit of this for us is exactly the modularity, which gives the ability to *change* a service with a defined API by any service exposing the same API and logic [143]. Thus, it allows the use of the same service located on another site, or of another version of a service (newer or older), or the same service in the same location but on a different server (for load-balancing purposes), or even an entirely different service, as long as all these services expose the same API and have the same logic.

For example, a load balancer makes good use of this property to distribute the load between multiple instances of the same modular service [81]. Figure 8.3 shows this process with a reverse proxy for load balancing lb_t of the service t . In the figure, lb_t intercepts and

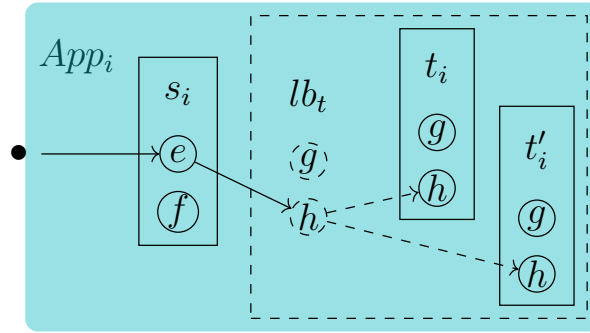
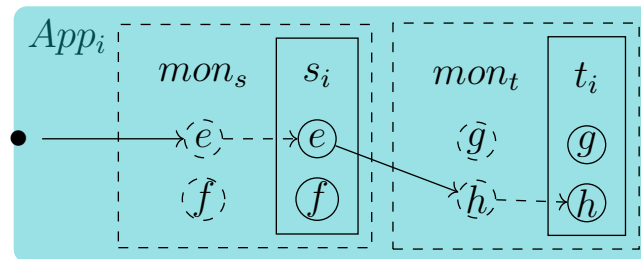


Figure 8.3 – Load balancing principle

balances incoming requests within two service instances t_i and t'_i during the execution of the workflow $s.e \rightarrow t.h$ in App_i . From one execution to another, the endpoint $s_i.e$ gets result from $t_i.h$ or $t'_i.h$ in a safe and transparent manner thanks to the modularity.

Thus, we can redirect a request to another instance of the same service.

As microservices architectures are the foundations of [DevOps](#) practices, because their modular decomposition of services allows for their independent instantiation and maintenance. As discussed in [Section 5.4](#), a service mesh takes benefit from this decomposition/modularity to implement communication-derived features such as monitoring, message routing or load balancing outside of the application. In its most basic form, a service mesh consists in a set of reverse proxies around service instances, each proxy encapsulating a specific code to control communications between services [\[28\]](#). This encapsulation in each proxy *decouples* the code managed by [DevOps](#) for their infrastructure deployment from the application business logic maintained by programmers. It also makes the service mesh *generic* to all services by considering them as black boxes and only taking into account their communication requests.

Figure 8.4 – Service mesh *mon* for the monitoring of requests

[Figure 8.4](#) illustrates a service mesh monitoring requests to have insight about the



application. The reverse proxies mon_s and mon_t collect metrics on requests directed to service instances respectively s_i and t_i , in this case, during the execution of the workflow $s.e \rightarrow t.h$ on App_i . They are illustrated as dashed rectangles around the services and a mimic of the endpoints present in the services. The encapsulated code in mon_s and mon_t collects for example requests latency and success/error rates. It may send metrics to a time series database as InfluxDB [144] and could be changed by anything else without touching any of the App application code.

The collaborations we need between multiple instances of an application can actually be done by using a service mesh on the infrastructure and programming redirections of the workflow. Indeed, a service mesh could allow dynamic composition of services on different sites. And because a service mesh is generic, it can be extended to all applications built by service composition.

At this point, with the modularity of services and the service mesh logic, we have the ability to use another instance of a service and a way to do it easily and dynamically.

To program dynamically the collaborations, we developed a domain specific language called *scope-lang*.

8.2 Scope-Lang, a DSL to reify locations and collaborations of requests

To define dynamically how a particular request should be handled by the service mesh, we developed a Domain Specific Language (DSL) that extends the default API with location information. Entitled *scope-lang*, it enables users to specify, for each resource, in which context the execution should take place.

Scope-lang extends the way clients and users interact with the services, whether it is through command lines or through a REST API. Because we were originally working on OpenStack, the *scope-lang* was designed to be added at the end of command lines; but it could obviously be included in a HTTP request, like in the header.

A *scope-lang* expression (referred to as the *scope* or σ in Figure 8.5) contains location information that defines, for each service involved in a workflow, in which instance the execution takes place. To put it simply: considering the previous application, the scope

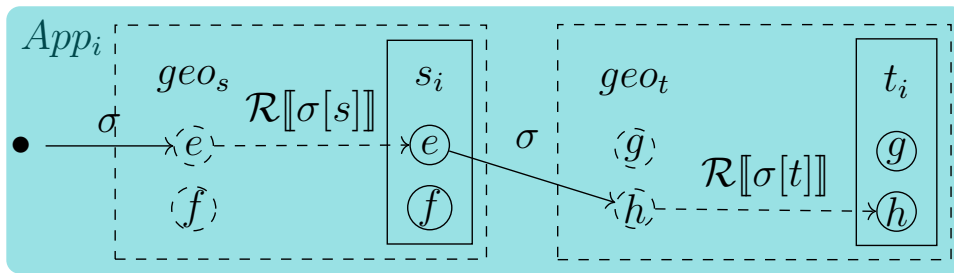
“ $s : App_1, t : App_2$ ” intuitively tells to use the service s from App_1 and t from App_2 . Conversely, the scope “ $t : App_1 \& App_2$ ” specifies to use the service t from App_1 and App_2 .

App_i, App_j	$::=$	application instance
s, t	$::=$	service
s_i, t_j	$::=$	service instance
Loc	$::=$	App_i single location
	$ $	$Loc \& Loc$ multiple locations
	$ $	$Loc \% Loc$ cross locations
σ	$::=$	$s : Loc, \sigma$ scope
	$ $	$s : Loc$

$$\begin{aligned}
 \mathcal{R}[\![s : App_i]\!] &= s_i \\
 \mathcal{R}[\![s : Loc \& Loc']]\!] &= \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']]\!] \\
 \mathcal{R}[\![s : Loc \% Loc']]\!] &= \mathcal{R}[\![s : Loc]\!] \text{ spread to } \mathcal{R}[\![s : Loc']]\!]
 \end{aligned}$$

Figure 8.5 – scope-lang expressions σ and the function that resolves service instance from elements of the scope \mathcal{R} .

Figure 8.5 presents the logic of scope-lang as a grammar and the function \mathcal{R} that resolves the workflow induced by the given scope. We will discuss later (Section 8.3) the different opportunities of collaborations offered by scope-lang. For now, we will mention only that the comma “,” represents the *sharing* collaboration; the ampersand “&” represents the *replication* collaboration; and the percent “%” represents the *cross* collaboration. These are the three main types of collaborations we envisioned, with more operators that could be defined for more flexibility in the requests.



$$\sigma = s : App_i, t : App_i$$

Figure 8.6 – Scope σ interpreted by the geo-distribution service mesh geo during the execution of the $s.e \xrightarrow{\sigma} t.h$ workflow in App_i . Reverse proxies perform requests forwarding based on the scope and the \mathcal{R} function.

Clients set the scope of a request to also specify the collaboration between instances



they want for a specific execution. The scope is then *interpreted* by our service mesh during the execution of the workflow to fulfill that collaboration. The main operation it performs is *request forwarding*.

Figure 8.6 presents, as before, an application App_i with two services s_i and t_i , but this time they are encapsulated, behind proxies. These proxies in front of service instances (geo_s and geo_t in Figure 8.6) the requests and execute them according to the defined scope. “Where” the requests will be executed exactly depends on locations in the scope.

The interpretation of the scope always follow these steps:

1. A request is sent to the endpoint of a service of one application instance. The request piggybacks a scope, typically as an HTTP header in a RESTful application. For example in Figure 8.6: $\bullet \xrightarrow{s:App_i, t:App_i} s.e.$
2. The reverse proxy in front of the service instance intercepts the request and reads the scope. In Figure 8.6: geo_s intercepts the request and reads σ which is equal to $s : App_i, t : App_i$.
3. The reverse proxy extracts the location assigned to its service from the scope. In Figure 8.6: geo_s extracts the location assigned to s from σ . This operation, noted $\sigma[s]$, returns App_i .
4. The reverse proxy uses a specific function \mathcal{R} (see Figure 8.5) to resolve the service instance at the assigned location. \mathcal{R} uses an internal registry. Building the registry is a common pattern in service mesh using a *service discovery* [28] and therefore is not presented here. In Figure 8.6: $\mathcal{R}[s : \sigma[s]]$ reduces to $\mathcal{R}[s : App_i]$ and is resolved to service instance s_i .
5. The reverse proxy *forwards* the request to the endpoint of the resolved service instance. In Figure 8.6: geo_s forwards the request to $s_i.e$.

In this example of executing the workflow $s.e \xrightarrow{\sigma} t.h$, the endpoint $s_i.e$ has in turn to contact the endpoint h of service t . The reverse proxy geo_s propagates the scope on the outgoing request towards the service t . The request then goes through stages 2 to 5 on behalf of the reverse proxy geo_t . It results in a forwarding to the endpoint $\mathcal{R}[t : \sigma[t]].h$ that is resolved to $t_i.h$. Here, the scope only refers to one location (i.e., App_i). Thus the execution of the workflow remains *local* to that location. The next section details the use of forwarding in order to perform collaborations between instances.

8.3 What kind of collaborations do we need?

Previously, we presented how a load balancer changes the composition between multiple instances of the same service *inside* a single application instance. In contrast, in the previous example for scope-lang, we change the composition between multiple instances of the same service *across* application instances. As a consequence, these different service instances on different sites can share their resources during the execution of a workflow.

This is done only by redirecting requests to other application instances, located on other sites. We call this mechanism *forwarding*, and it is the basis of all the possible collaborations. We will now see different collaborations envisioned, of which the first three are essential to provide a single coherent system (sharing, replication, cross, mentioned before in Section 8.2), and the others only adding some possibilities for the users.

8.3.1 Forwarding for resource sharing

Figure 8.7 depicts the dynamic composition mechanism described above during the execution of the workflow $s.e \xrightarrow{s:App_1, t:App_2} t.h$. The service instance s_1 of App_1 is dynamically composed thanks to the forwarding operation of the service mesh with the service instance t_2 of App_2 . This forwarding is safely relying on the guaranty provided by modularity: if t is modular, then we can swap t_1 by t_2 since they have the same API and logic, as they are different instances of the same service. This implies obviously that they have to be the same major version to present the same API.

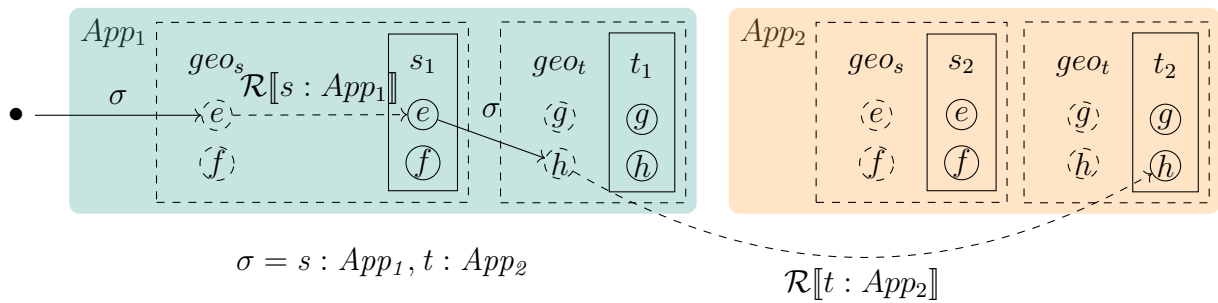


Figure 8.7 – Resource sharing by forwarding between instances

The process is similar to the one described in Section 8.2, but with $s : App_1, t : App_2$, resulting in a forwarding to the endpoint $\mathcal{R}[t : \sigma[t]].h$ that is resolved to $t_2.h$, with t_2 the t_i service instantiation in App_2 . Since it is so similar, we avoid repeating it further. As a

result of this request, the endpoint $s_1.e$ benefits from resource values of $t_2.h$ instead of its usual $t_1.h$.

This is the collaboration we call *resource sharing*, or shorter *sharing*.

8.3.2 Forwarding for resource replication

Replication is the ability to create and maintain identical resources on different sites: an operation on one replica should be propagated to the other ones according to a certain consistency policy. In our context, it is used to deal with latency and availability.

Microservices often follow a RESTful HTTP API and so generate an identifier for each resource. This identifier is later used to retrieve, update or delete resources. In our approach, each application instance is independent. Therefore, it requires a meta-identifier so users can manipulate replicas across the different sites as a unique resource and to unify these identifiers. Furthermore, we need to associate this meta-identifier to local identifiers for each involved site. To do so, we propose a mapping $\{metaId : [App_i : localID_i], service_name\}$ to keep track of replicated resources and apply further changes to all replicas. The service name will be used locally by the registry of the service mesh to find which service to contact.

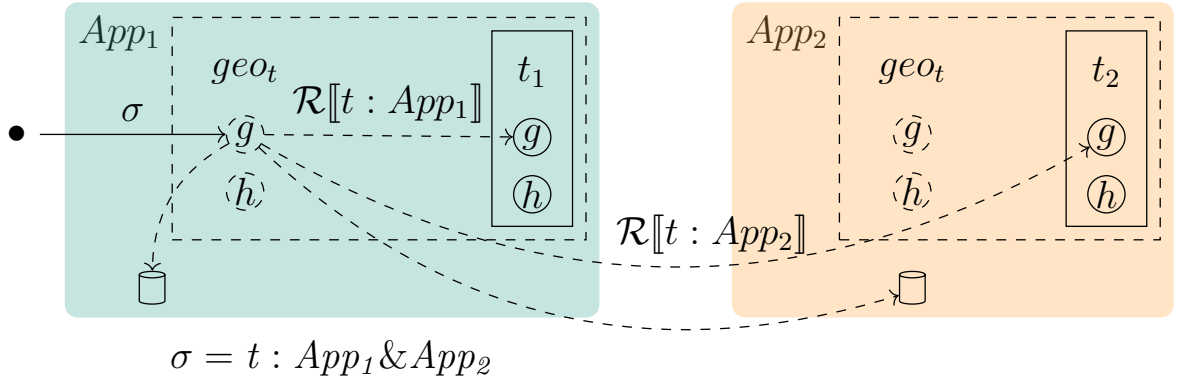


Figure 8.8 – Replication by forwarding on multiple instances

For example, in Figure 8.8, the service t exposes an endpoint g that creates a resource. When using a scope to create replicas of the same resource, such as $t : App_1 \& App_2$, the service mesh generates a meta-identifier and maps it $\{metaId : [App_1 : localID_{t_1}, App_2 : localID_{t_2}], t\}$. If t_1 creates a replica of the resource with the identifier 42 and t_2 with the identifier 6, and our meta-identifier was generated as 72, the mapping is: $\{72 : [App_1 : 42, App_2 : 6], t\}$. These mappings are stored in an independent database alongside each

application instance, in the control plane (see Section 5.4, Figure 5.3 for a reminder on the control plane of a service mesh).

The replication process follows these steps:

1. A request for replication is addressed to the endpoint of a service of one application instance. For example in Figure 8.8: $\bullet \xrightarrow{t:App_1 \& App_2} t.g$.
2. Similarly to the sharing, the \mathcal{R} function is used to resolve the endpoints that will store replicas. $\mathcal{R}[s : Loc \& Loc'] = \mathcal{R}[s : Loc]$ and $\mathcal{R}[s : Loc']$. In our example: $\mathcal{R}[t : App_1 \& App_2]$ is equivalent to $\mathcal{R}[t : App_1]$ and $\mathcal{R}[t : App_2]$. Consequently, t_1 and t_2 .
3. The meta-identifier is generated along with the mapping and added in the database. In Figure 8.8: $\{72 : [App_1 : none, App_2 : none], t\}$.
4. Each request is forwarded to the corresponding endpoints on involved sites and a copy of the mapping is stored in those sites' database simultaneously. In Figure 8.8: geo_t forwards the request to $t_1.g$ and $t_2.g$ and stores the mapping $\{72 : [App_1 : none, App_2 : none], t\}$ in App_1 and App_2 databases.
5. Each contacted service instance executes the request and returns the results (including the local identifier) to the service mesh. In Figure 8.8: t_1 and t_2 returns respectively the local identifier 42 and 6.
6. The service mesh completes the mapping and populates the involved sites' databases. In Figure 8.8: the mapping now is $\{72 : [App_1 : 6, App_2 : 42], t\}$ and added to databases on App_1 and App_2 sites.
7. By default, the meta identifier is returned as the final response. If a response other than an identifier is expected, the first received response is transferred (since others are replicas with similar values).

This process ensures that only interactions with the involved sites occur, avoiding wasteful communications. Each operation that would later modify or delete one of the replicas will be applied to every other using the mapping available on each site. To prevent any direct manipulation that would break the consistency, either local identifiers of replicas can be hidden to the users or each operations executed need to check in the database if the local identifier exist in a mapping, but it is really slower. Another way is to warn users from the beginning that if they manipulate replicated resources through their local identifiers, no guarantees can be offered on the consistency.



This replication control perfectly suits our collaborative-then principle. It allows a client to choose “when” and “where” to replicate. Regarding the “how”, our current process for forwarding replica requests, maintaining mappings and ensuring that operations done on one replica are applied on others, is naive. Implementing advanced strategies is left for the implementation. However, we underline that it does not change the foundations of our proposal. Ultimately, choosing the strategy should be made possible at the scope-lang level (e.g., weak, eventual or strong consistency), through different operators, for example.

Finally, to conclude on the approach of the different collaborations and the scope, we have to add that in the absence of a scope for an execution, every operation in the workflow will be local to where the request was made. Moreover, if all the involved services are not specified in the scope, because a lot of services can be involved in a single workflow, the execution will be local to where the request is currently located. This keeps the approach local-first as much as possible.

8.3.3 Forwarding for cross

This last collaboration is the less mature one and another PhD candidate, Geo Johns Antony is working currently on it specifically.

Cross is inspired from an early study that focused on extending virtual networks between multiple Edge sites [7]. The idea is to create a resource over multiple sites. The main difference to the aforementioned replication collaboration is related to the aggregation/-divisibility property. A cross resource can be seen as an aggregation of all resources or part of resources that constitutes the cross-resource overall. Some resources which cannot be divided by an application API will require an additional layer in the business logic to satisfy the divisibility property.

Cross thus revolves around two properties:

Aggregation This property aggregates the resources from various involved geo-distributed sites. This property is maintained at each of these locations. Cross is aware about each involved site. It provides the illusion of a single site for resources which are geo-distributed across sites. Users instantiate the request with scope-lang and Cross aggregates this request to give results from multiple geo-distributed sites.

Divisibility Cross divides the resources into various geo-distributed locations. While creating a resource, if the resource is further divisible by the application API, cross



manages to divide the resource into further smaller resources. While dividing, cross creates a prime site, which will be the site of the location in the request and other sites get an illusion of the resource from this site. For each resource, the prime site may be different, as it is defined per resource by the users.

A **CREATE** operation for instance can distribute the resource over different sites (if this resource is divisible), while a **GET** will be performed on each "sub-resource" composing the cross-resource in order to return the aggregated result. Similarly to the replicant data scheme, the different resources involved are monitored in order to perform CRUD operations in the expected manner.

The illusion of single-site is created with two primary principles. At each location, all the resources are available and extended. Any CRUD operations can be performed at any of the sites specified with the scope and applied to the entire Cross resource. The interaction will be similar to performing a request local to the application. The communication layer created by our generic solution ensures the transfer of request across each site.

How we deal with the split brain issue for cross-resource is left as future work. However, it is worth noting that the unreachability of one site that hosts a part of the cross-resource faces multiple challenges. These challenges are currently addressed by Geo Johns Antony, in his own PhD thesis.

8.3.4 More (and more about) collaborations

We discussed in this manuscript about three collaborations, namely, sharing, replication and cross. But to fit perfectly to the needs of the users, we can also envision more types of collaborations and operators. The reflection presented in this section has been presented in our Euro-Par paper [\[31\]](#).

The code of scope-lang is independent of cloud applications and can easily be extended with new features in it. Scope-lang is thus a great way to implement additional operators to give more control during the manipulation of resources, although it has been initially designed for resources sharing and replication, and then cross. Choosing between different levels of consistency in the replication is one example of possible new operators. For some resources, applications can require stronger or weaker consistencies, depending on what is done with these resources. Each consistency could co-exist with different logics and operators according to the resources requirements. This would allow the users to define



what type of consistency they need for different resources, and thus for example when operations should be fast and when they can be slow to improve consistency, such as a RedBlue consistency [145].

In this section, I present two other operators to stress the generality of the approach regarding possible operations, namely the *otherwise* operator and the one derived from it, the *around* operator.

Otherwise

The new *otherwise* operator (“ $Loc_1; Loc_2$ ” in Figure 8.9) informally tells to use the first location or fallback on the second one if there is a problem. This operator comes in handy when a client wants to deal with sites disconnections. Adding it to the service mesh implies to implement in the Cheops core what to do when it interprets a scope with a (;). The implementation is straightforward: make the service mesh forward the request to the first location and proceed if it succeeds, or forward the request to the second location otherwise (if it fails for any reason).

$$\begin{array}{ll}
 Loc & ::= \dots \quad (\text{see Figure 8.5 for } \dots) \\
 & | \quad Loc; Loc \quad \text{otherwise location}
 \end{array}$$

$$\mathcal{R}[s : Loc_1; Loc_2] = \mathcal{R}[s : Loc_1] \text{ otherwise } \mathcal{R}[s : Loc_2]$$

Figure 8.9 – The otherwise (;) operator.

Around

Ultimately, we can built new operators upon existing ones. This is the case of the *around* function that considers all locations reachable in a certain amount of time, e.g., `around(App1, 10ms)`. To achieve this, the function combines the available locations with the *otherwise* operator (;), as shown in Figure 8.10. This is made possible through the heartbeats between different instances of Cheops that also retrieve the latency between the sites. Thus it does not require to change the code of the interpreter in the service mesh, only a small addition (presented in Figure 8.10).



```
def around(loc: Loc, radius: timedelta) -> Loc:
    # Find all Locs in the `radius` of `loc`
    # > locs = [App1, App2, ..., Appn]
    locs = _find_locs(loc, radius)

    # Combine all `locs` with `;`
    # > App1;App2;...;Appn
    return foldl(;;, locs, loc)
```

Figure 8.10 – The `around` operator build upon `(;)`.

Compositions of collaborations

Another element we did not discuss entirely in this manuscript is *the composition of collaborations*. If we consider for example, as suggested in the classification thereafter (Section 8.4), the replication of a resource using a sub-resource on one site, we combine replication and sharing. Such a request would follow a command such as, for the creation of two replicas of a resource *a* on *Site₁* and *Site₂*, with a sub-resource from *Site₁*:

```
application create a --name bar --sub-resource foo \
    --scope{Service A: Site_1 & Site_2, Service B: Site_1}
```

Except for the dealing of dependencies (it would be probably better to replicate also the sub-resource, if they are linked by reliance), this is pretty straightforward. This composition should be done in one step, as opposed to the one for reliance, where the users would first need to replicate the sub-resource (*foo*) before using it for the creation of *bar* replicas (and avoid using sharing).

Some compositions can be more tricky even. What happens when users want to create a sub-resource inside a cross resource which will not allow the name of the replicas to co-exist, as the creation of pods through replication inside a cross namespace that exist on several sites on Kubernetes? The namespace spans on different sites, and we want to create a pod *foo*, replicated of those different sites. Unfortunately, the namespace will not accept the local creation requests of *foo*, as the name will exist after the first request. The problem with that is that it is not really a replication problem, because replication will only execute the request without the scope locally on each sites, and it is not exactly a cross problem, as Kubernetes manage replicaset by giving replicas unique names.

This means that we need dedicated code to manage the composition of collaborations, mainly because of the dependencies.

8.4 Classification of resources dependencies

The classification of resources dependencies is a joint, on-going work with Geo Johns Antony, who defined a draft model which was then refined together, and adapted for our own collaborations. This has been presented in our ICSOC 2022 paper [35, 138]. Also, once again, the focus is made on the replication, but there is similar work for cross.

Many resources have dependencies on each other (a virtual machine in the OpenStack ecosystem depends on an image, a network, an IP, etc.; a deployment file in Kubernetes is linked to several pods; etc.). In fact, the sharing collaboration implies there is some kind of dependency between the involved resources. Hence, it is mandatory to rely on a relationship model for replication and cross operations. This model will be used to keep track on each critical resource and ensure that CRUD operations are performed thoroughly.

We have identified and formulated three types of dependencies, depicted in Figure 8.11.

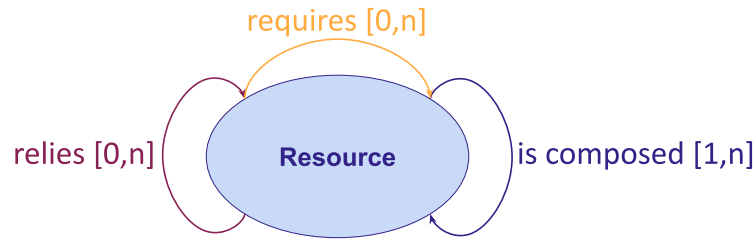


Figure 8.11 – The different dependencies

Requirement

Requirement defines a relationship between two resources that is not critical for the survival of either of the resource but rather is a necessary link during a particular operation. The operation can be any operation performed upon either of the resource. While performing the operation the link is vital and if the link is severed the resultant operation will terminate and it will not succeed. If the link is maintained and no external factors affect the operation, the operation will be a success and after this the link between these resources is insignificant. Hence, a broken link after the operation does not affect either of the resource. An example for OpenStack is a VM requires an image, for the creation operation.



Reliance

Reliance defines a relationship between two resources that is critical for the survival of either one of the resource or both. If the link between these resources is cut at some point during the lifetime of these resources it will impact the existence of the resources and can lead to a failure condition. Involved resources are independent and one resource cannot alter the other resource. For example, in Kubernetes, a pod, when created with a secret, relies on this secret.

Composition

Composition consists of intrinsic dependencies between resources: the life cycle of the two resources are linked. The creation of resource A implies the creation (and respectively the destruction) of resource B. Composition is obviously wider than just two resources as one resource can be linked to a collection of other resources, which in their turn can also depend on sub-resources. For example, in OpenStack a stack can be composed of VMs, and in Kubernetes, a deployment is composed of pods.

8.4.1 Creation patterns for replication operations

As mentioned, the goal of the relationship model is to ensure that Cheops operations are done thoroughly when the manipulated resource is not elementary, but depends on other resources. We discuss in this paragraph the various cases.

Requirement For a replication scenario, the user have the choice to first replicate B everywhere A will be; in this case, the creation of A can be executed without specifying the location of B, it will be executed locally on each site. The other choice is to specify the dependency in the creation request, which is represented in [Figure 8.12](#).

1. Using the sharing operator in the scope, the user specifies that a resource B required is on *Site1*.
2. Cheops intercepts the request to get a resource from another site when it will be sent by the service needing it.
3. Cheops transfers the request to get resource B from *Site1*.
4. Resource B is received and the usual flow is executed.
5. Since Resource B is only required for some operations, this dependency is stored in Cheops database for further usage (in these operations).

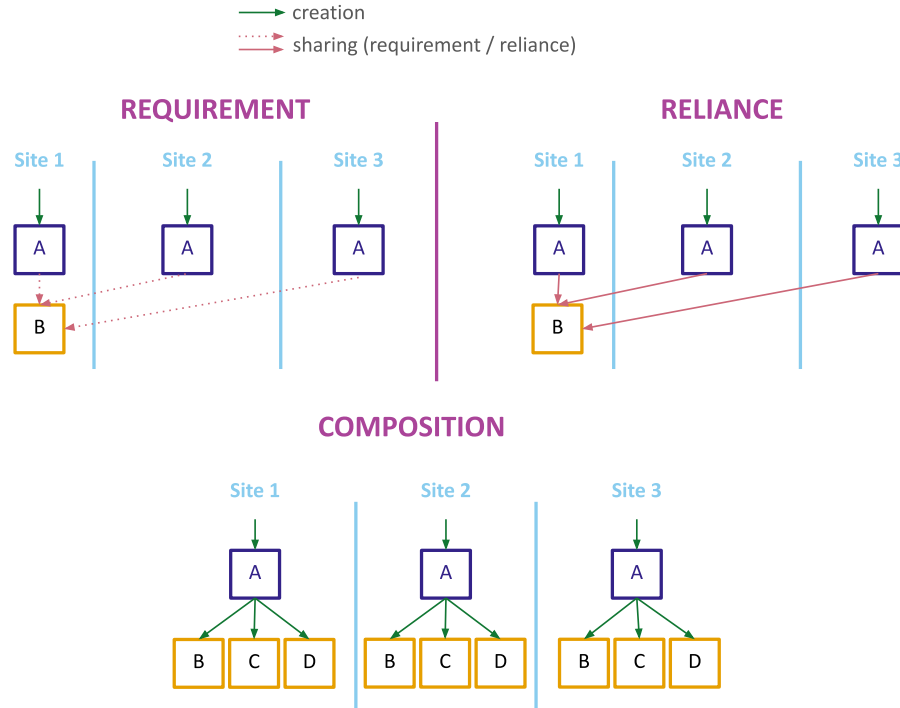


Figure 8.12 – Behaviors to observe following the dependencies

Reliance This relationship follows a similar approach from requirement for replication. The difference is with the involvement of Resource B through the life span of Resource A. At any point if Resource B fails for both collaborations, resource A will result in a failure state. The primary objective being to preserve the strong relationship between Resources A and B, Cheops needs to ensure the reachability of Resource B.

As before, a user can still replicate Resource B to ensure that Resource A will not suffer from a network partition. Otherwise, the process is the same as the requirement, except for: first, the dependency information needs to be stored in Cheops database for resource B to warn users against resources failures in replicas in case of a deletion of B. Second, Cheops needs to warn the users of affected resources (replicas of resource A) in case of network partition that affects B, because they will be in a failure state.

Composition For replication scenario, a copy of resource A is created on the involved sites which in turn creates resources B, C and D on each of these sites with a *cascading effect* from the normal, local execution of the creation of A. An update



on resource A for a secondary layer resource B, C or D is propagated across the involved sites and also follows normal execution on each site. Network split brain is managed through the Raft protocol that ensures eventual consistency between all replicas.

Though this classification is still mainly theoretical, with some implementations being currently studied, the implementation of the rest of the approach will be presented in the next chapter.

CHEOPS, OUR SOLUTION TO PUSH CLOUD APPLICATIONS TO THE EDGE



In this chapter, I will present the implementation of the global approach, and what we achieved to have our service mesh dedicated with the purpose of pushing applications to the Edge by managing their geo-distribution. This service mesh is called Cheops.

9.1 Towards a full implementation of our approach in Cheops

In the appendix ([Chapter 12](#)), I present the first version of the implementation, that was a first step towards our current approach. In this section, we will discuss the current implementation, which is still under development.

Cheops is a service mesh, with a proxy intercepting the request for each service as the data plane. The control plane is deployed on each site, as a service, as it is one more service of the application instance. In the rest of the manuscript, I will refer to “Cheops” as the Cheops service deployed on each site; they all communicate between each other to allow for collaboration, forming a service mesh with the proxies. Each of them maintains a catalog of the service instances on its site to know how to contact them locally. Each Cheops knows its neighbours and contact only the other Cheops involved in a request, and they are responsible to contact the involved services with their local catalog. More importantly, to keep it generic, Cheops considers each type of resources as black boxes.

In the next figures, we do not present the specific involved endpoints, as the important part is the services involved. Each service is shown as a rectangle, named as *ServiceX_i* with *ServiceX* the name of the service and *i* the identifier on the site where it is located. The proxies are represented as an oval around the services. The Cheops databases are represented only when they are necessary. Manipulated, involved resources are pictured

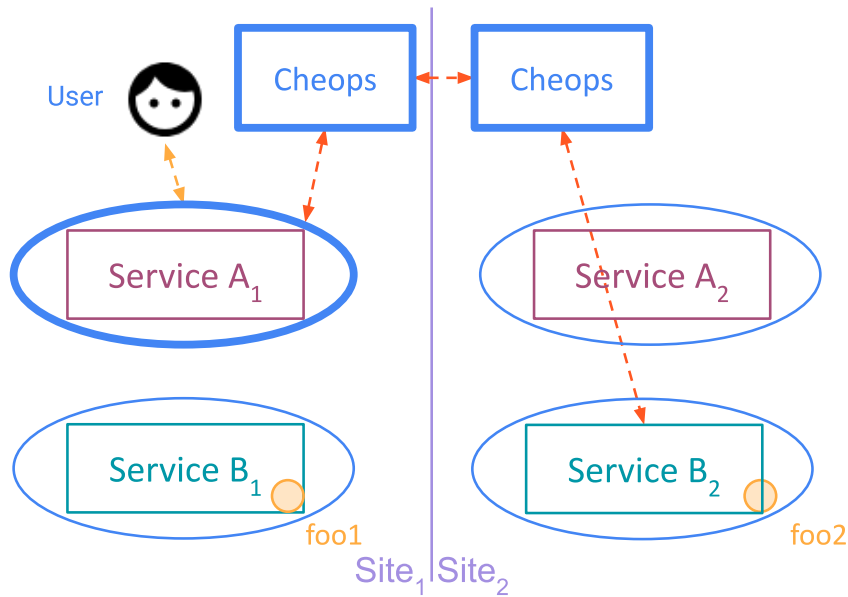


Figure 9.1 – Sharing uses services instances from different sites

as a small circle on each service.

9.1.1 Implementation of collaborations

We will first detail how the three collaborations were implemented in *Cheops*, or at least some overview on how they should be implemented.

Sharing

Sharing was a joint work envisioned by Ronan-Alexandre Cherrueau, Adrien Lebre, Matthieu Simonin and Javier Rojas Balderrama with a PoC named OpenStackoid [146, 136, 96].

As explained in the previous chapter, *sharing* is the collaboration which allows a service instance to use a resource from a service which is not the one assigned to its application instance. This is the first way of leveraging the entire infrastructure, by distributing different resources on different sites and allowing their use from one site on another, enforcing the collaborative-then principle. It actually also enforces the local-first principle, because it allows the execution locally, using only resources from other sites when required.

The typical example is getting a resource from a Service B on another site for a Service A, the workflow presented as red arrows in Figure 9.1. For example, if *ServiceA* requires



a *foo* resource from *ServiceB*, the command line given by users to request the use of resource *foo2* from *ServiceB*₂ on *ServiceA*₁ is:

```
application create a --name bar --sub-resource foo2 --scope {A: Site1, B: Site2}
```

1. A user requests to create a resource *a* on *ServiceA* from *Site*₁ (Service *A*₁), using a sub-resource *foo2* from *ServiceB* on *Site*₂ (Service *B*₂).
2. The request is intercepted and transferred to Cheops (control plane).
3. Cheops extracts the scope from the request and interprets it, in this case, learning that it should be directed to the local *ServiceA*.
4. Cheops transfers the request to *ServiceA*, that executes it until it needs the sub-resource from *ServiceB*.
5. The outgoing request is intercepted, and at this point, is transferred to *Site*₂ for forwarding on *ServiceB*₂.
6. Cheops on *Site*₂ uses its catalog to find Service B endpoint locally and transfers the request to get *foo2*.
7. The service response (containing the resource itself) is finally transferred back to Service A through Cheops.

Similarly to a local failure, if *ServiceB*₂ is not reachable, the request cannot be performed.

We can see here that contrary to the intra-services collaboration we talked about in [Section 3.3](#) that requires dedicated code, our approach use directly the response from another service instance, which is offered by default.

Replication

We will give here a first overview on how replication can be implemented, a detailed version will be given later on, as it was the specific collaboration I studied in our approach, and here, we give an overview for all collaborations. *Replication* is the ability for users to create and have available resources on different Edge sites to deal with latency and split networks. Replication main action is duplication: transfer the request to every involved site and let the application execute the request locally.

The operation does not simply consists though in forwarding the request to the different instances. Cheops keeps track of the different replicas in order to ensure that future

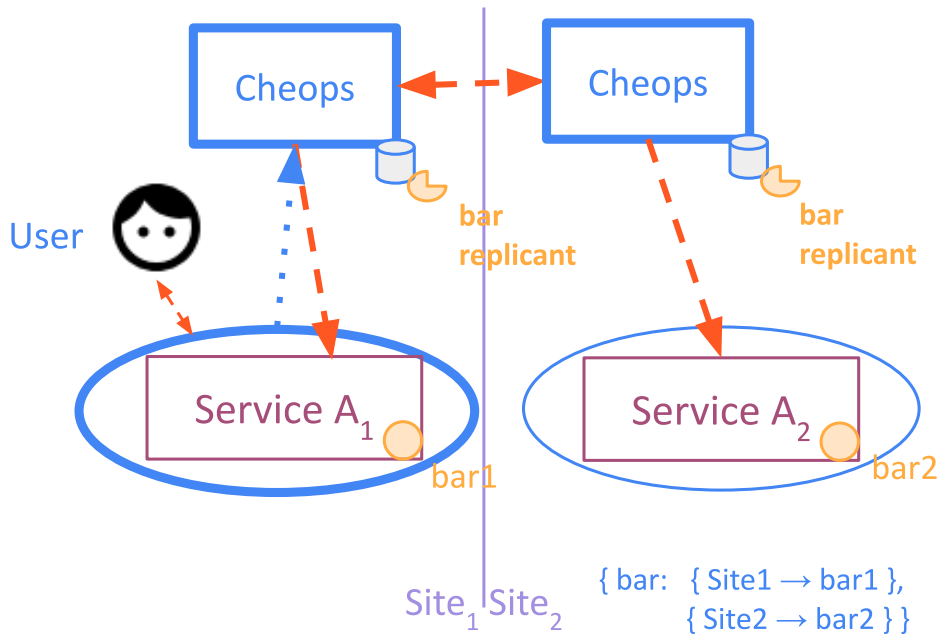


Figure 9.2 – Replication executes an operation on multiple instances

CRUD operations achieved on any replica will be applied on all copies, maintaining eventually the consistency over time from the CRUD point of view. To do that, Cheops relies on a data scheme, called the replicant, that links a meta-ID to the different replica IDs and their locations (called the mapping in the previous chapter). Replicants are located on every Cheops database on the sites involved in the request to replicate, just as the replicas are located in every application database on these same sites.

Figure 9.2 sums up the workflow to create a replicated resource on two sites:

`application create a --name bar --scope {Service A: Site1 & Site2}.`

1. A user sends a request on Site₁ to create two replicas of the *bar* resource on *Service A* from Site₁ and Site₂, i.e., *ServiceA₁* and *ServiceA₂*.
2. The request is intercepted and transferred to the local Cheops (control plane).
3. Cheops extracts the scope and interprets it, in this case, it needs to contact *ServiceA₁* and *Cheops* from Site₂.
4. Cheops creates the replicant, and passes the request to create it to Cheops on the other involved site (Site₂).
5. Both Cheops execute the request of creation of *bar* locally, which is simply the request without the scope; the response is intercepted to fill the local IDs on the



replicants and the response is transferred to the user, replacing the local ID by the meta-ID of the replicant.

To provide eventual consistency, Cheops follows the Raft protocol, with one replicant acting as the leader. As for the other collaborations, it is crucial to understand that since we manipulate the resources through the API their services expose, we can only intervene from this point of view.

Cross

As mentioned earlier, Cross is not entirely mature, but has been implemented in part in Cheops, and thus it is worth developing a bit more.

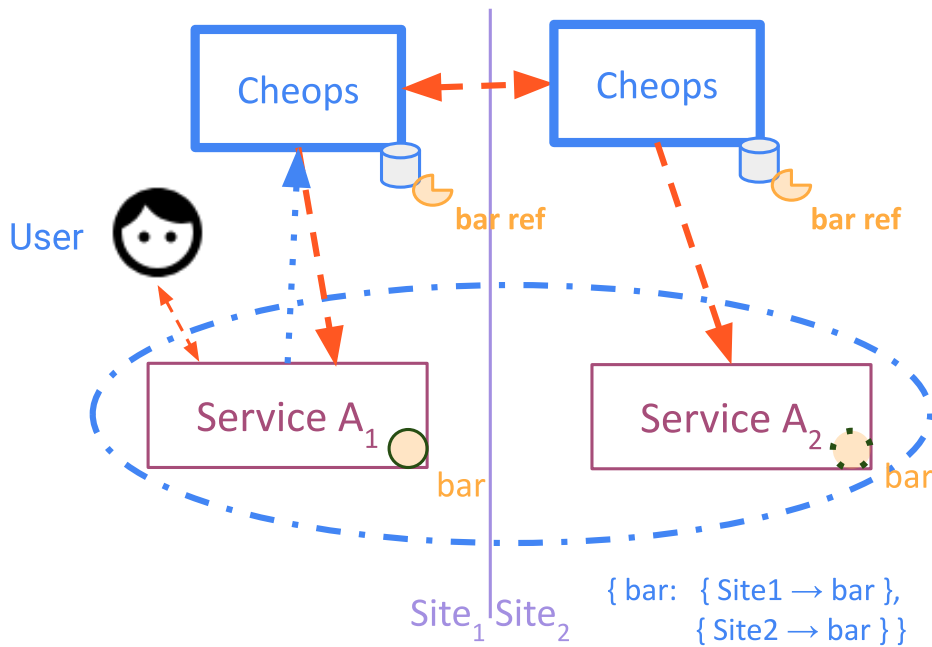


Figure 9.3 – Cross gives the illusion that multiple resources behave as a single one.

An illustration of Cross is depicted in Figure 9.3, following this request:
`application create a -name bar -scope {Service A: Site1 % Site2}.`

1. A user sends a request to create a resource specifying the involved sites in the Cross operation.
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope and interprets it.

4. Cheops creates the resource on the first site ($Site_1$) and passes the request to Cheops of other involved sites (here, $Site_2$ only).
5. Cheops on $Site_2$ identifies the extended resource and creates an identifier within Cheops to forward to the deployed resource site.

9.1.2 Cheops architecture

The global architecture of our proof-of-concept is depicted in Figure 9.4. Cheops follows a modular approach, composed of various microservices, which are linked together through REST API protocol. There is one Cheops per site and they monitor known Cheops through heartbeats.

For redirection purposes, each Cheops has its own internal registry. First, it stores information about its own site: the addresses of each application service endpoint as a catalog or registry. Second, it keeps the addresses of its Cheops neighbours to know where to transfer the requests when needed. Cheops never intercepts requests coming from a Cheops to an application service to avoid looping.

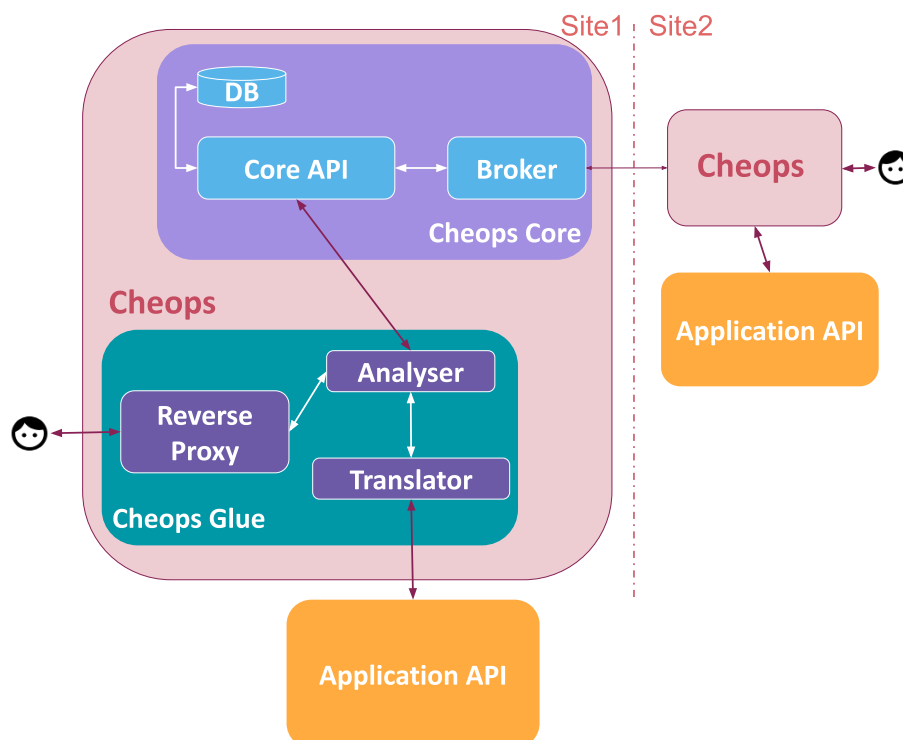


Figure 9.4 – Cheops architecture



Cheops internals

The Cheops control plane is divided into two main components which are Cheops Core API and Cheops Glue.

Cheops Core is the primary building block of Cheops. It encapsulates the communication module, interface module (Core API) and the database. The communication module provides the link between multiple Cheops instances, thus creating and maintaining a service mesh around the various involved clusters. Since Cheops focuses on minimizing intrusive code changes of a deployed application, this module plays a significant role, while addressing the collaboration features of Cheops. This module also communicates with the the Core API module in the Core. The Core API is a management module created to interconnect all the services inside Cheops.

Cheops Glue is the second module of Cheops. This module is designed to help Cheops Core translate Cheops API requests into the respective application API and vice-versa. Core is designed to handle agnostic API requests which are irrespective to all the applications. Cheops needs to convert these requests into application understandable API patterns. Since each application has its own pattern for intercepting API, Glue is developed with respect to individual applications such as Openstack, Kubernetes, etc. Thus, this module is not generic to every approach and has to be implemented to translate the requests, but still allows the externalization of geo-distribution concerns. Glue acts as a first layer communication between users and Cheops by intercepting the data from the default CLI from the respective application using scope-lang. The analyzer service in Glue evaluates the request and converts it into a generic request understandable by the Cheops Core API service. On the other side, Cheops Glue provides a translator service which converts the requests received from the Cheops Core API service. Cheops Glue will also manage the creation of extra business logic for divisibility property of cross collaboration (see [Figure 9.3](#)) for specific types of resource. The Glue also handles the network requirements and implements the relationship model (that we will only discuss in the future work part of the conclusion).

The design of Cheops architecture is focused on a modular design which makes it easier to manage and distribute. It also provides the flexibility of enhancing each component or individual service to scale it during peak usages. It is available for any request specifying the usage of its hosting site when it is not disconnected from the network (not offline)

and the users are also able to control this instance locally, making it available at any time. Cheops obviously offers the three collaborations proposed earlier. It manages each collaboration without the need to modify anything in the deployed application. Finally, it maintains the consistency of the deployed applications across the locations.

9.2 A deeper dive into the replication

As previously introduced, replication is the ability to create and maintain identical resources on different sites: an operation on one replica should be propagated to the others, dealing with faults and disconnections and maintaining CRUD consistency based on our eventual model. Other consistency policies [147, 148] could be envisioned, but let us leave that as future work as they do not change the general concept of scope-lang/Cheops. To get a better understanding of the point of replication, imagine a user who needs a huge resource (like an ISO image) both at home and at work. The resource can be replicated at creation on both sites and it will be the only time when the entire resource will go through the network. This saves a lot of bandwidth, and is especially useful if there is a partition between both sites, or if the user wants to work offline.

9.2.1 Replication model

A lot of this model has been discussed in [subsection 8.3.2](#) and [Section 9.1.1](#), so we will avoid as much as possible repetitions, unless it can be explained differently for a better understanding.

A replicant is basically a meta-identifier we generate along with a list of mappings $site \rightarrow local_identifier$ and a service name (or identifier). A replicant can thus be implemented for example as:

$$meta_identifier : [site_n : local_identifier_n, \dots], service_name$$

It is important to know that for now, the service name has not been used in the current implementation as the request given has been enough to transfer to the correct service.

These replicants are stored in a database co-located to the Cheops agents. A copy of the replicant is stored on each site where its replicas are located (the sites involved in the replication). Cheops has an API of its own to allow the user to check the state of operations, sites and inspect replicants.



9.2.2 CRUD execution workflow

First, to define what is the creation, update or delete workflow, we have to define what they do in our consistency model and what are their boundaries.

In replication, resources are only considered as black boxes seen only as the API allows it, and thus the consistency maintained only by the operations allowed by the API (usually CRUD operations). Any modification made internally on one site by the application without using the API cannot be expected to be replicated to the other replicas. Furthermore, any modification made on those replicas through the API will be applied eventually to the other replicas.

The creation of resources replicated in an eventual consistency implies that all replicas are identical at creation and will be created eventually. The update of resources created with the replication in an eventual consistency implies that all replicas will be updated eventually, whether the user specifies a scope or not in its request. It is the same as updates for deletes.

About the boundaries, the operation obviously begins when the user makes the request. But for the end, we could consider that an operation ends either when there is one response and is returned to the user, or when the operation is executed on every site. In an eventual consistency model, the latter (the operation done on every site) can come a lot later than the first response.

It is important to know what happens in case of failure (partition, disconnection, server failure) during the execution until the first response, but also after, because the operation must be executed on our replicas at some point.

In terms of the user view, the first response to arrive goes to the user before it might be applied everywhere, so it is the responsibility of the user to check with a request to Cheops or directly to involved services and sites to know where the creation or updates are already applied. Users cannot assume because they received the answer the operation as already been applied everywhere, some sites may have difficulty to execute the request or may be even down.

Creation

The replication process to create a resource on App_1 and App_2 has already been detailed in [subsection 8.3.2](#) and [Section 9.1.1](#). Therefore, we will not develop it further in this part, and the reader can refer to these parts to know about it.



Read

Since the replicas are the same from the API level view, on a probability of a replica not updated yet, read one replica or another should not be different, so only one is needed to be returned. Therefore, the process of reads is straightforward; to access a specific resource, users must either request their site where one of its replicas is (local-first) or specify in the scope on which location a replica of the resource to read is (collaborative when needed).

Update

After the creation of replicas, every request made to update (or delete) is filtered to check if the id given corresponds either to a replicant meta identifier or a local replica identifier. Once again, this process is really long, so it is pretty costly, but it ensures the consistency of the approach. Ideally, it should be configurable, so the users can take responsibility for the divergence of state in the replicas if they use local replicas identifiers and everything is not analyzed. The process is quite similar to the creation, but does not generate a new replicant or change an existing one. It only applies an update to replicas and update the logs of replicants.

1. A request for an update of a previously created replica is addressed to the endpoint of a service of one application instance.
2. Cheops checks if the ID in the request exists in the replicant database. If not, the request is sent back to the service to be executed. If it is, the request is transferred to the Cheops agent storing the replicant leader. It gets the corresponding replicant to find all replicas (and thus sites) involved. The leader sends a vote to the replicants to have a consensus on the request. When it gets the consensus, the operation is stored in its log.
3. The request is copied as many times as necessary (with the corresponding local identifier from the mapping so it will work properly on the different involved sites) and sent to the Cheops agent of involved sites.
4. Local Cheops agents send the request to the corresponding service on their site, which executes the request normally.
5. Each Cheops agent sends back the response to the Cheops agent where the replicant leader is.



6. This agent sends back the first response to the user, once again, with the meta-identifier where the local-identifier would be expected to notify the user that the replicas were updated.

Delete

As for the update, a delete on replicas can be identified either by a local identifier or the meta identifier. The process is identical as the update's.

It is important in this subsection that adding replicas to a set of pre-existing replicas situations have not been fully studied for implementation yet, though they have been considered. Conversely, the removal of a replicant from the set is important in the processes of dealing with faults.

9.2.3 Dealing with faults

We define a fault as: a partition of network on an involved site (disconnection), or a failure from this site, whether it is shut down, out of order, or if the request cannot be executed for any reason (not enough memory to create a resource for example).

It is also important to mention that if the site where the user sent its request is faulty (does not work in any of the aforementioned way), the request obviously cannot be executed. The user can make the request to a more distant site, but this is one of the advantage of replication.

Moreover, the “during an operation” can refer to two distinct phases. As we discussed before, the end of an operation can be seen as: when a replica has been created/updated/deleted and the user has been notified, and when the operation is applied to all replicas. So “during an operation” is between the request of the user and before one of these ends. In our consistency model, this conveys no difference to the process.

If a site fails where a replica is supposed to be, other Cheops will be informed due to its heartbeat (or rather lack of). Any other operation received by the leader will then be retried according to the log when the site comes back again. Therefore, a site is considered to be eventually available again unless it is removed. If a site is removed from the system, every site that was hosting a replica must delete the site from its mappings (from the replicant). The leader will be in charge of this particular task, sending a request to update the replicant.

Faults during operations The operation will be applied *eventually* on all involved

sites. This eventual consistency uses a consensus protocol, and in our case, an implementation of Raft [149]. For example, the leader’s log allows to replay operations that are not yet applied. It is the responsibility of Cheops hosting the leader to ensure that operations are applied eventually, by checking regularly operations that have not been yet applied.

Faults while there are replicas but not particular operation When a site fails while there are replicas somewhere without any particular operation running, no heart-beat is received by other Cheops agent and the replica is considered unavailable temporarily.

If a site where a replica is was partitioned at some point but could be used locally, only read queries can be made, and these reads might be stale (not be up-to-date to the operations that have been made on other replicas). When rejoining the cluster, operations will be applied on the site so it is up-to-date thanks to the leader’s log.

9.3 Testing Cheops replication

We demonstrated the feasibility of our proposal on the Kubernetes ecosystem. The feasibility for the collaborations were studied for replication, in which we manipulated replicated pods across two sites.

The experiments have been performed over two sites of the Grid’5000 experimental testbed [150], as it was used for the first version of Cheops (see in the appendix, [Chapter 12](#)). The sites were completely independent of each other and located on two sites of the infrastructure, namely Nantes and Rennes. On each site, we deployed a Kubernetes cluster, each composed of one master and one worker node, as well as Cheops.

9.3.1 Current technology stack

Cheops was deployed onto each site alongside an instance of the application (Kubernetes), creating a [P2P](#) service mesh structure between these applications. The primary goal was to create an initial development efforts as per the architecture [Figure 9.4](#). These efforts provided the functionalities we proposed to create a P2P service mesh, integrate collaboration mechanisms and flexibility to extend to multiple applications.

For the initial study, Cheops was built on Golang, since it is very adaptable to server-side scripting. In order to create the service mesh, Cheops relies on a message broker



which provided P2P capabilities and a reliable messaging protocol. Advanced messaging queuing protocol (AMQP) was chosen for our design of the Cheops-to-Cheops communication (communication module of the Core). RabbitMQ¹ with P2P and AMQP messaging pattern were used for this purpose.

For the database, the main requirement was to find a free and open source database (for easier adoption) which could replicate the data based on individual attributes. After a survey, the conclusion made was that no particular database existed which met all the required characteristics, and thus ArangoDB² was chosen to be integrated into Cheops, to allow us to have a NoSQL document store that could keep different types of data.

The next component needed was the one that can intercept the users requests in order to identify the request patterns and especially to be able to get the scope in requests. HAProxy³ was chosen to provide a lot of flexibility in order to integrate between the application and Cheops. *Nonetheless, it is crucial to know that though we deploy HAProxy in the current PoC, we do not intercept requests and we only use Cheops API to make the request.*

In more details for the replication, Figure 9.5 presents the model for the Replicant structure, which is composed partly of a slice (re-sizable array) of Replica objects. In the Replica struct, there are information on the Site (name in our case, but it could be an ID of a site for very large scale) on which the Replica is, the ID of the local resource, and a Status that is not used for now, but was added as a maybe simpler way to know if one replica is reachable or not. In the Replicant struct, we store the Meta-ID as expected, the different Replicas (from the above struct), a Boolean to know if this particular Replicant is the leader of the Replicants, and finally the Logs of the different operations executed on the Replicas to maintain consistency. For those who are not used to Golang, the json information at the end of the fields of the structures is for encoding and decoding to and from JSON, so they can be passed with a REST request.

9.3.2 Experiments

The artifact we used for the experiments is available on the Cheops project page⁴ (around 3k lines of code). It is a joint work from Geo Johns Antony and myself.

-
1. <https://www.rabbitmq.com/> - Accessed 2022-10-09
 2. <https://www.arangodb.com/community-server/> - Accessed 2022-10-09
 3. <https://www.haproxy.com/> - Accessed 2022-10-09
 4. <https://gitlab.inria.fr/discovery/cheops/>

```

type Replica struct {
    Site    string `json:"Site"`
    ID      string `json:"ID"`
    Status  string `json:"Status"`
}

type Replicant struct {
    MetaID    string    `json:"MetaID"`
    Replicas  []Replica `json:"Replicas"`
    IsLeader  bool      `json:"IsLeader"`
    Logs      []Log     `json:"Logs"`
}

```

Figure 9.5 – Actual replicant model in Cheops code.

The goal of the experiments we performed was mainly to validate the expected behavior and genericity on simple requests. Further experiments are still needed, and as mentioned, some experiments on cross have also been performed but will not be presented here.

Table 9.1 presents the commands tested and their results.

Operation	Location	Result
<code>kubectl create pod purple --scope{Site1&Site2}</code>	Site1	Pod <i>purple</i> created on Site1 and Site2
<code>kubectl create pod violet --scope{Site1&Site2}</code>	Site2	Pod <i>violet</i> created on Site1 and Site2
<code>kubectl get pod violet</code>	Site1	Pod <i>violet</i> from Site1 is displayed
<code>kubectl get pod violet</code>	Site2	Pod <i>violet</i> from Site2 is displayed

Table 9.1 – Experimentations on Kubernetes - replication.

We created one pod *purple* and one pod *violet* respectively on **Site1** and **Site2**, with the scope specifying to create those pods as replicas on both sites. This means we should have a *purple* on **Site1** and **Site2**, and the same for the pod *violet*. This corresponds to the commands `kubectl create pod purple --scope{Site1&Site2}` and



`kubectl create pod violet --scope{Site1&Site2}`, respectively on Site1 and Site2.

Then we requested the pods *violet* from each site to check if they were present. This corresponds to the commands `kubectl get pod violet` on both Site1 and Site2.

Though this set of requests tests really basic functionalities, we were able to ensure that the *create* and *get* operations were working as intended.

Additional experiments are under progress to test *update* and *delete*, as well as scenarios on the behavior of replicas in case of network split to check the consistency. A study on different types of resources to better check the genericity is also required. Especially, we used Kubernetes for this, because the approach has already been tested on OpenStack, but we did not try this particular PoC, so this could be a further important test.

SUMMARY OF THE APPROACH



To fulfill the requirements stated in [Section 3.1](#), using different existing technologies and means, Cheops and scope-lang are a solution to bring existing, service-based Cloud applications to the Edge. The main goal is allow any of these applications to be run on Edge infrastructures without changed in their code, by taking into account the properties of the Edge.

Cheops a service-mesh which main functionality is to manage the geo-distribution of the Edge sites and the resources of the applications. Scope-lang allows users to define the execution location and collaboration in their requests **dynamically, on-demand**.

With the application deployed on each site, the solution answer the **local-first** requirement automatically. The different collaborations, of which we presented three, namely sharing, replication and cross, fulfill the **collaborative-then** requirement.

By externalizing the geo-distribution concerns, the approach is **non-intrusive**, and it is as **generic** as possible by only considering resources as black boxes and using the application API, with only the Cheops glue specific to the applications.

The Cheops service mesh functions in a **decentralized** and **P2P** manner, and is designed with autonomous sites and some collaborations, to be **resilient to network partitions**.

Conclusion





Christie: How many [...] are there?

Dr. Wren: Twelve.

Alien: Resurrection

DISCUSSION



This chapter is dedicated to discussions around the approach, its inherent limitations and what could be done better or in the future. There is an entire chapter ([Chapter 11](#)) for the perspectives on the approach, but in this chapter, we will only talk about small changes that could be done in the Cheops implementation and would not fundamentally change the approach.

The first small point to discuss is the naming “service mesh” itself. The approach is called service mesh because of the definition of William Morgan [Section 5.4](#) (page 51). It is a layer handling service-to-service communication that delivers request through the topology of services of cloud native applications, implemented as an array of proxy alongside the services, and the application is not aware. In Cheops, the only functionality is to help the application running at the Edge by managing the geo-distribution of instances of this application. But Cheops also fulfills some of the functionalities of an API gateway, if we consider this definition from IBM [\[151\]](#):

An application programming interface (API) gateway is software that takes an application user’s request, routes it to one or more backend services, gathers the appropriate data and delivers it to the user in a single, combined package. It also provides analytics, layers of threat protection and other security for the application.

An API gateway provides a single entry point for all API calls that come into an application, whether the app is hosted in an on-premises data center or on the cloud. It accepts requests that come in remotely and returns the requested data.

The functionalities of analytics, threat protection and security are definitely not considered, but what is done is Cheops could be viewed as a decentralized API gateway (with several entry points that are only the same Cheops instantiated on every site). Overall, the name “service mesh” has been chosen to help grasping the idea of what we do more easily, but we have functionalities from different ways to manage services from Cloud applications.



Our approach, as it has been said, takes into account different interesting features or way to approach problems that have been presented in the state-of-the-art, [Part II](#). The closest approach is [\[152\]](#), which uses some kind of sharing collaboration for Glance, tracks ID of the resources, and has autonomous instances of OpenStack on each site. Cheops differs mainly by being more generic and using different types of collaboration to fulfill the single coherent view.

10.1 Application version

As discussed before in [Section 3.1](#), the approach relies on the modularity of service-based Cloud applications, so it is not viable for any type of Cloud applications. It requires microservices-based applications that exposes an API for services communications. These applications also need to be able to work on a single site since they will be deployed autonomously on every sites. Moreover, it has been mentioned shortly, but every instance of the services should have the same version so they have identical models of their resources and they have the same API.

This can be restrictive as it is difficult for different actors on the entire infrastructure to maintain the same version, as it has been seen with OpenStack, different businesses often have their own version of the application and have different versions of the application, some dating from several years; and it is actually the case sometimes in the same actor with different locations¹. This is mainly because it is difficult to update a running application, but also because it is difficult for open-source applications to be tailored for every use case possible.

10.2 Consistency

10.2.1 Consistency from the API

This approach ensures consistency at the service-level, but for the resources they manage. The only operations available to manipulate these resources are therefore the ones exposed by the API. Thus, the resources are maintained as identical as the API allows it, but nothing more and any change on the resources that can be done internally

1. <https://cleura.com/services/cloud-features/regions-and-services/> - Accessed 2022-10-12



make the replicas diverge from others. For example, in the OpenStack case, VMs created through replication will have the same specifications, but everything done internally on the VMs will not be replicated to other VMs as the application API does not allow to make these changes.

It is a limitation of the approach, but this consistency *inside* the resources is not desired as it requires to know the internals of the application and reproduce it outside or add code inside the application to provide an API.

10.2.2 Better consistency

Raft is a distributed consensus algorithm using an elected leader to apply the changes made on replicas. The leader sends changes to the followers and await confirmation of the majority of followers to confirm the change before committing it. In our case, since we cannot always assume it is possible to rollback a change in a resource, we only ensure that a majority of nodes are available before sending the request to be executed. Thus, the consistency is only ensured eventually, when the changes will be executed on all involved replicas.

As another consensus algorithm, Epaxos could be considered, to avoid designated leaders [153, 154].

The use of Conflict-free Replicated Data Types (CRDT) [155, 156] is also currently under study for future versions of Cheops.

10.3 What can be improved in Cheops

First of all, it is important to mention once again (see [Section 9.3](#)) that in our current implementation of Cheops, in order to simplify the process, we decided to avoid the interception part as it is mostly technical. Nonetheless, it should be considered for the validity of the approach as it could be a problem for some applications, such as Kubernetes, as Cheops acts as a man in the middle attack. Fixing these security concerns from Kubernetes is a problem for the security, but also for the approach, because we would be meddling in the business code. On the other side, it is possible to monkey patch the Kubernetes API as it has been done for OpenStack in OpenStackoïd [136]. In any case, we are still missing this part for Cheops, currently.

From HYDRA's paper [76], we could also implement the decision of a new leader to



be on the closest site in terms of latency to the one that is currently unavailable. This is debatable though, as the closest site from the previous leader is not necessarily the closest to the user. This brings the question of whether users should be able to decide where the leader of replicants should be, in case of mobility, to lower the latency for requests to the leader. These are points to be considered for the future.

TOWARDS A GENERALIZED CONTROL SYSTEM: FUTURE WORK



11.1 Mid-term deliveries for Cheops

From what has been discussed in the previous chapter ([Chapter 10](#)) comes a lot of work to consider for Cheops to be thought and/or done sooner rather than later.

Moreover, there is still work to consider from the theoretical views we have on Cheops, such as more collaborations ([subsection 8.3.4](#)) and the classification of resources ([Section 8.4](#)). The classification in particular is under heavy discussions currently, and an answer on how to use it with a dependency tree has been studied thoroughly in the PhD work of Geo Johns Antony.

11.2 Ownership Types to prevent meaningless collaborations: the long haul

This perspective has been the main focus of my thesis for some time and was not explored further after it has been set on the side to focus on the global approach and replication. Most of this section comes from our research report [\[96\]](#) and has not been peer reviewed. This work comes in part from the observation we made from the exploration of using geo-distributed databases to push Cloud applications to the Edge [\[25\]](#) that sharing some resources make no sense.

As said many times before, the scope, in our proposition, is built by users according to their needs. But this human defined scope is error prone, first of all because of potential typing errors, but moreover, users might not be well informed of the availability of a resource. The problem arises when sharing resources is that some sharing are fallacious because *resources exist in a specific scope*.



To better illustrate this, we use an example scenario from OpenStack: “Boot of a VM attached to a flat network”. In OpenStack, a flat network is often used to attach a VM to an existing physical network, and the command would look like (omitting some specifications from other services):

```
openstack server create my-vm --network flat
```

Figure 11.1 depicts how a request is made on only Site 1 with the “192.168.0.0/16” physical network. After the user issued a boot request (step 1), the compute service requests for a port on the flat network (step 2). A port is a specific resource containing the information to attach the VM to the network. It includes a mac and IP address, henceforth the user can reach the VM according to them at the end of the boot (step 3). In a local setup, everything works as expected: the VM is reachable at “192.168.0.3”, because this network has been set up on this site.

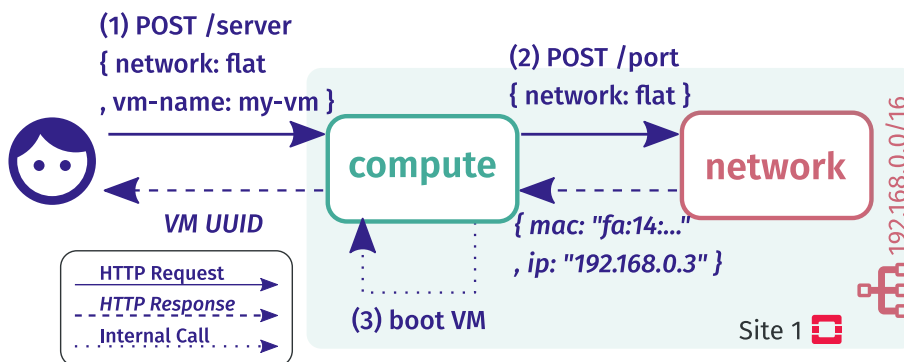


Figure 11.1 – Boot of a VM attached to a flat network locally

Now, if we consider two different sites, with Site 2 also having a physical flat network, but on a different domain (i.e., “10.0.0.0/8”). Therefore a problem may occur if the user wants to use the share operation such as “boot a VM on Site 1 using the flat network from Site 2”. This operation can be expressed in scope-lang with the following command:

```
openstack server create my-vm --network flat \
--scope { compute: Site1, network: Site2 }
```

Such an operation makes no sense, as it is presented in Figure 11.2. Site 2 returns a port for its physical network (step 2 response) as it usually would. It includes the IP address “10.0.0.2” that is going to be used by the VM in Site 1 (step 3). Unfortunately, Site 1 has a different physical network (“192.168.0.0/16”). Therefore, though this operation is executed successfully, the VM created with this address becomes unreachable.

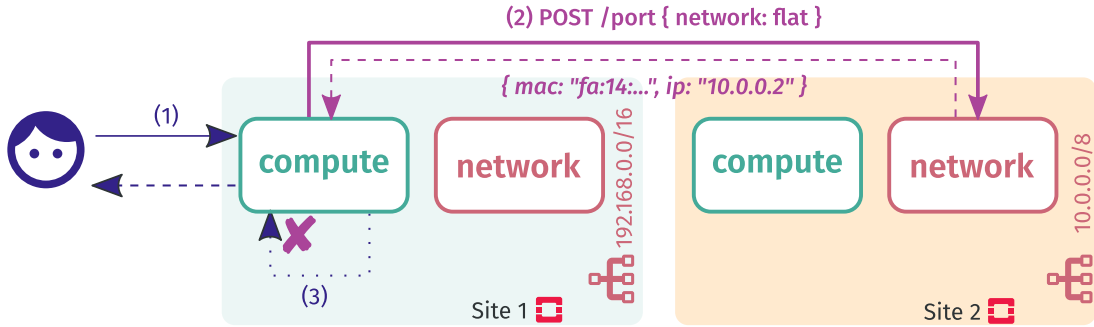


Figure 11.2 – Launching a VM with a network, using collaboration

Similarly to name bindings in programming languages, we say that resources have a scope that defines their *visibility*. This scope could be local to one application instance (e.g., in our example, a physical network should only be visible by its local site). Or it could be global to all application instances (e.g., in OpenStack, an image is visible by all instances).

Sharing a resource with a local scope such as in Figure 11.2 is a problem. Here, the VM is created with *side effects* that are associated with this creation. These effects are costly because they use multiple resources while being of no use (since the VM will be unreachable). More importantly, it is impossible to define a general rollback strategy that undo all these side effects. So it is crucial that each resource sharing is validated before its execution.

A naive approach to identify invalid resource sharing would be to exhaustively list all correct ones. While it is a pragmatic approach for a specific use case, it does not offer extensibility and lacks of generality. This approach could work for one given version of an application, but any changes or additional features would result in invalidating the aforementioned list.

To avoid such dependencies to one specific application, we propose to leverage memory access control techniques. In programming, a particular information has a validity in its own memory context. This information can be used in different contexts as long as there is a *reference* that enables its access. Sharing and copying this reference into other pointers leads to a well-known issue called pointers aliasing (e.g., dangling pointers). An simple example of a dangling pointer is presented in Figure 11.3. A pointer `ptr1` references an object by its address in the memory. At some point, a copy of the pointer (`ptr2`) is made, for example for a shallow copy of the object. The object goes out of scope, either because it was a local variable and the function is done, or because a user deletes it. The memory is de-allocated, but `ptr2` still points to it.

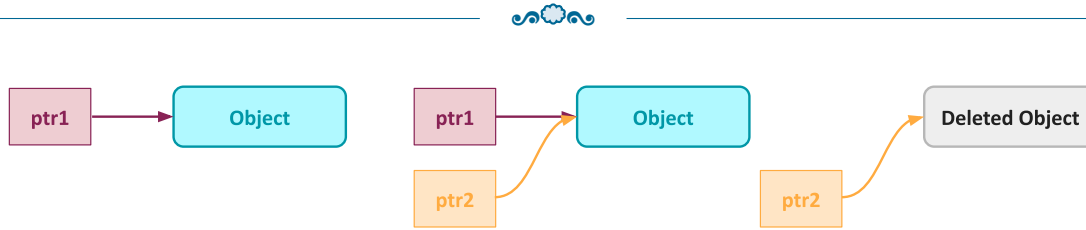


Figure 11.3 – Dangling pointer example.

```
class Network<m>: # Network service
    def getFlatNetwork() -> m/FlatNetwork: #...
class Compute<m>: # Compute service
    def createVM(network: m/FlatNetwork)-> m/VM: #...

# Code of the OpenStack CLI for
# > openstack server create --network flat
network: Site2/Network = Network<Site2>()
compute: Site1/Compute = Compute<Site1>()

flat_net: Site2/FlatNetwork = network.getFlatNetwork()
compute.createVM(flat_net)
#           ^ mismatched types:
#           `createVM` expects `Site1/FlatNetwork`,
#           but found `Site2/FlatNetwork`.
```

Figure 11.4 – Ownership types (in purple) to prevent invalid collaborations

Among the solutions that have been proposed to deal with this issue, the ownership types (OT) [157, 158] has defined the notion of containment. At coarse-grained the idea is to only allow the owner of the memory space to access it, preventing references sharing and copying.

We propose to use in the future this concept in scope-lang to prevent wrong collaborations. In the network scenario, Site 2 owns a network resource but the user shares its reference with Site 1. It resembles the dangling pointer problem: the scope of the network is Site 2 and so cannot be reached by Site 1. The sharing of this resource must be prevented.

Using OT, it becomes possible to specify the scope of this resource (i.e., a type that defines which OpenStack instances is its owner). This type can be used in a type-checker to prevent any wrong sharing a priori. The pseudo-code in Figure 11.4 shows the OT annotations and type-checking of our network scenario. We first define the services (l. 1-4). Then instantiate them with Compute in Site 1 and Network in Site 2 (l. 6-9). Finally, we create a VM on Site 1 using a reference to the network from Site 2 (l. 11-12). Here, the type-checker complains about the ownership of the network.



The OT proposal should not need changes in the business code of the application. However, it requires to type services API in addition to implement a type-checker.

This ownership types approach should be developed further to help ensure that sharing of resources will be done in a manner that prevents non valid sharing.

This approach is a long haul to consider and implement and thus is considered for the distant future of Cheops.

CONCLUSION



The Edge computing paradigm has shifted the way applications need to be designed to cope with a hostile environment where disconnections are the norm rather than the exception. With the need for low latency and robustness against disconnections, it is difficult to envision using existing applications designed for the Cloud at the Edge.

To run such an application on such a widely spread infrastructure, it is crucial to consider scalability as well as locality and tolerance to faults in the infrastructure.

In this thesis, I have studied how it is possible to avoid creating new applications specifically for the Edge, but rather use existing, service-based Cloud applications.

Summary of the approach

To deal with the latency and disconnections, the application used need to be deployed entirely on each Edge location, which allows for a local-first approach, where the application work autonomously on each location. Then, to allow for a single coherent system cherished in distributed systems, for mobility and the usage of the entire infrastructure, it is mandatory to give the application the ability to be collaborative-then. To allow for site collaborations, we need to be able to manipulate the location of request executions.

As Cloud applications can be huge, the solution needs to be generic and non-intrusive to avoid treating location information in the business code, so it is necessary to externalize the geo-distribution concerns outside of the application. Finally, because the Edge infrastructure is highly dynamic (with sites disconnections and failures) and to allow users to decide the location of the request execution (e.g., for privacy), it is important to give them the ability to choose it on-demand, dynamically.

The study of the state-of-the-art gave good hints on how to make all of these requirements happen in a P2P, fully decentralized manner, though it did not provide an entire solution for all of them.

A service-mesh dedicated to the management of the geo-distribution and collaborations



between applications was the solution that fit all the requirements to put existing Cloud applications at the Edge. To allow this approach, scope-lang is the DSL that supports the user-defined, on-demand, fine-grained request descriptions. The modularity of the service-based Cloud applications and the way their services communicate with each other through REST APIs are the major elements that helped our approach, by allowing the forwarding of requests, on top of which different collaborations are possible.

The current version grants three types of collaborations: sharing, that uses a resource from another site, replication, which allows users to put identical resources on different sites, and ensures that they will stay identical from the API point of view, and cross, which allows resources spanning across different sites. These three collaborations enable the use of resources locally first, and across sites when needed. They help to lower latency, allows redundancy, fault tolerance, and along with the users defined requests, they give the users the ability to choose at fine-grain where their requests will be executed and which resources to use, which also ensure their privacy, as they can select what sites they trust. In particular, the replication uses well known algorithms and logic to ensure consistency and tolerance to fault partition and it allows users to manipulate whichever replicas is closer/available.

As perspectives to improve this work, I gave hints of possible extension of collaborations thanks to the genericity of the approach regarding resources. I also presented the classification of dependencies to ensure the proper manipulation of linked resources. Finally, for the long haul, I explained how to prevent non-valid sharings through the use of ownership types.

Overview of the parts and chapters

This manuscript was built as this following description:

The introduction asserts the problematic of this manuscript. To be more descriptive, the goal is mainly to answer the question: is it possible to bring Cloud applications to the Edge? And with even more specifics, can it be applied to Cloud infrastructures management applications, and could we use a service-mesh approach to do it?

Part I explains the context of the manuscript.

Chapter 1 described the Cloud overall to give the reader a context of what the Cloud is. It gave a view of how Cloud applications function and how to manage



the Cloud infrastructures to give a basis to understand the approach presented in this thesis.

Chapter 2 presented the Edge and its challenges to explain on which expectations we were going to build our approach. We then defined the principles which we think are necessary to follow when developing or adapting an application in the context of the Edge.

Chapter 3 described in more details how it is possible to adapt a Cloud application for the Edge and why the required collaborations are not good solutions for me as they imply really intrusive changes.

Part II depicts the State-of-the-art regarding the management of applications at the Edge, whether they were natively built for it or brought from the Cloud.

Chapter 4 and **Chapter 7** serve as introduction and conclusion for the State of the Art. The first present how we evaluated the literature and the latter present the overall comparison.

Chapter 5 presented six approaches to manage the infrastructure and more importantly, applications on top of it. It also presented in [Section 5.4](#) to of the most known service meshes to better understand how they function and how they behave regarding our requirements.

Chapter 6 showed two different approaches to develop an Edge-native application or adapt an existing one with intrusive changes.

Part III introduces my approach to bring Cloud applications to the Edge using a service mesh like approach.

Chapter 8 explained our own theoretical approach to respond to the expectation we presented in [Section 3.1](#).

Chapter 9 presented in more detail the PoC we develop to correspond to the aforementioned approach, in particular the way the replication is implemented and how it was tested.

The conclusion chapters above presented discussions about our approach, its limitations and different perspectives to improve it.

Appendix



FROM CONSUL TO CHEOPS: FIRST TESTS ON A DUMMY CLOUD APPLICATION



During 2021 spring, I supervised two interns, Matthieu Juzdzewski and Arnaud Szymanek, worked on a first adaptation of our approach with the goal of giving information on using a service mesh to implement the theoretical approach given in [Chapter 8](#).

In this first version, the idea was to have an entire instance of an application on each site we want, and create a Cheops agent that would also stand on each site. These agents would be responsible to interpret the requests and transfer them according to the scope-lang. Also on each site, a reverse proxy besides every service, transferring their requests to the local Cheops agent for interpretation. This implementation used Consul service mesh¹ and Envoy² as reverse proxy to intercept and redirect, when needed, the requests.

The starting point for this work was we needed three components:

- intercepting/forwarding requests
- knowledge of the services of the target application and their endpoints
- extraction and interpretation of the scope in the requests

As it happens, a service mesh can execute the two first requirements, the last needing to be implemented ourselves because it is scope-lang specific. As we discussed in [Section 5.4](#), Envoy works as a sidecar proxy for every service and allowed for interception of requests. Consul also uses a catalog that can be used dynamically to register and unregister services³.

In this first version, there were two different components for Cheops: the *core* and the *connector*. The core received all requests to extract the scope, interpret where to send the request and forward it. If an address was local to the application instance (on the same site), it used the Consul catalog to get the corresponding service endpoint;

1. <https://www.consul.io/>

2. <https://www.envoyproxy.io/>

3. <https://developer.hashicorp.com/consul/docs/discovery/services> - Accessed 2022-09-25



otherwise, it was sent to the connector. The connector was responsible to send a request on a distant host by using a dictionary of known sites, and then the distant connector to the corresponding service on its own site.

Unfortunately, with the complexity of the service mesh and proxy, this work was working only on a simple use-case using two services (ServiceA and ServiceB), which only purpose was for the first to call for a resource on the second, to test if the sharing collaboration can be achieved. The original goal was to use Google Cloud Platform Online Boutique Demo⁴, but the purpose of this particular demo is to show the Anthos service mesh usage, and thus was not really designed to manipulate the resources inside. This is why we deferred to two simple services with basic functionalities and calls. Moreover, in this version of Cheops, Envoy was configured to only intercept and redirect requests for each service through a ServiceRouter configuration. Each service thus required a specific configuration to do so, and a specific endpoint in Cheops.

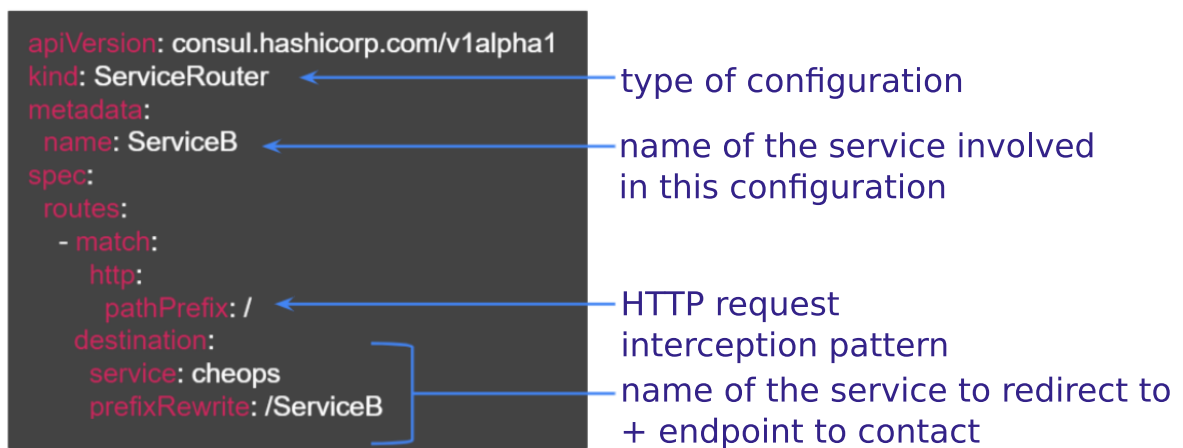


Figure 12.1 – Configuration of a service in Envoy in the first version of Cheops.

Figure 12.1 shows the type of configuration that was done to achieve the redirection to Cheops core. For each ServiceB (metadata → name), every request will be intercepted (the prefix is /), and will be redirected to the Cheops service, at the endpoint /ServiceB. Thus, Envoy will redirect indiscriminately every request that comes to ServiceB to Cheops for checking.

Figure 12.2 presents the workflow of a request to use *ServiceA₁* and *ServiceB₂*, which

4. <https://github.com/GoogleCloudPlatform/microservices-demo>

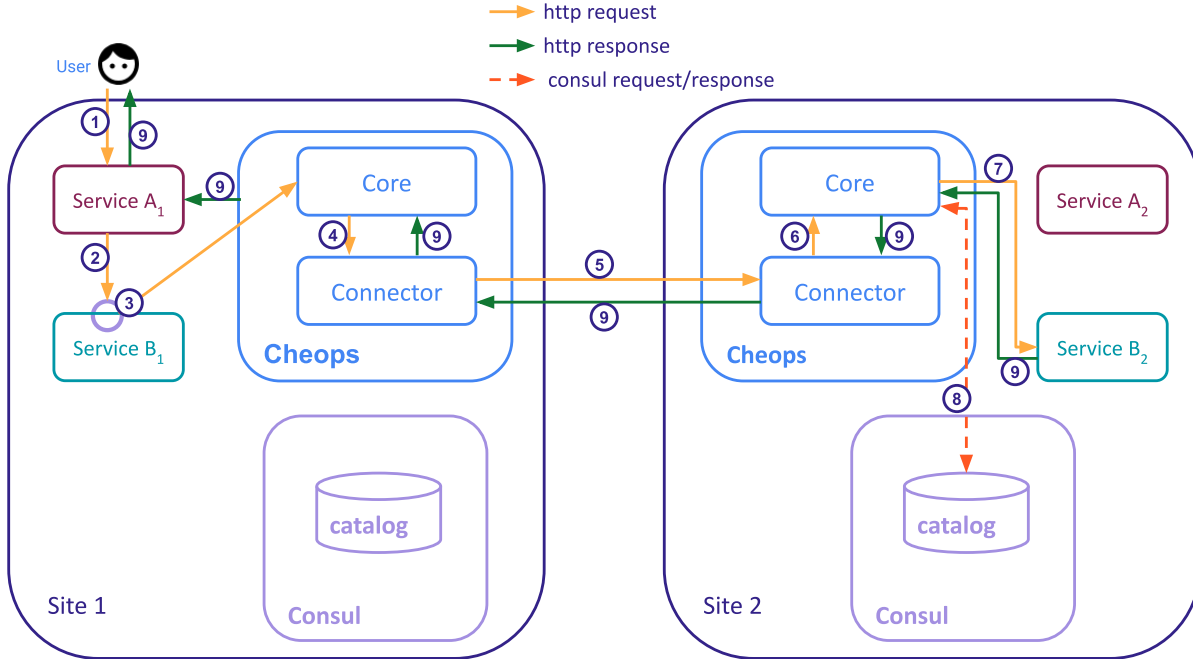


Figure 12.2 – Workflow of a sharing request in the first version of Cheops.

means to use ServiceA on Site 1 and ServiceB on Site 2. ServiceA has an endpoint e , which is called with an HTTP request: `http://address/ServiceA/e`. ServiceB has an endpoint h , which is also called with an HTTP request: `http://address/ServiceB/h`. This particular endpoint h is the one called by ServiceA to execute the workflow called with the endpoint e .

1. The request is made to use endpoint e of ServiceA such as
`curl http://address/ServiceA/e -H "scope: ServiceB/Site2"`, with the scope thus integrated in the header.
2. To complete the workflow, ServiceA contacts the h endpoint of ServiceB.
3. The prefix “/h” is recognized as an interception pattern for Envoy; the request is effectively intercepted and redirected to Cheops core on its /ServiceB endpoint.
4. The core extracts, interprets the scope and sends the request toward the connector since it cannot serve the request locally.
5. The connector finds in its registry the IP address for Site 2 and transfers the request to Cheops connector on Site 2.
6. The connector on Site 2 redirects the request to its local core.

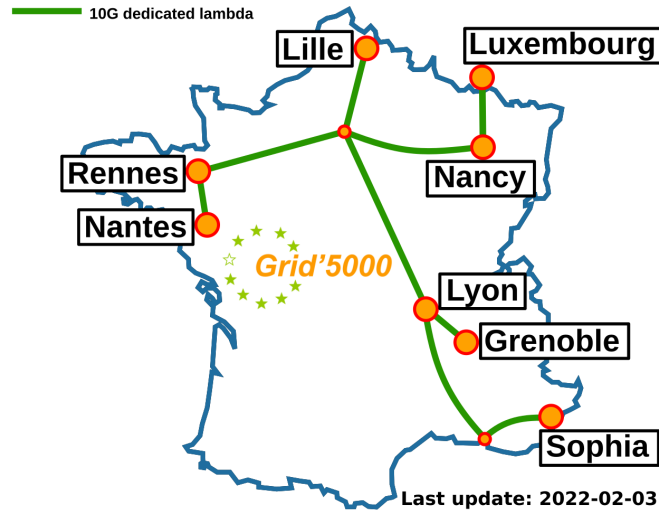


Figure 12.3 – Grid’5000 and the Renater network.

7. The core asks Consul catalog to get the address of the local *ServiceB* required; then finally transfers the request to this *ServiceB*₂.
8. The response from *ServiceB* is transferred back to *ServiceA*₁ where the resource will be used to complete the workflow.

This work on the first version of Cheops has been validated on two different sites (Nantes and Rennes as Site 1 and Site 2) on the Grid’5000 testbed [150] (see Figure 12.3). The services have been deployed as Docker Containers through Kubernetes.

The experiment only showed it is possible to use a resource from another site with sharing: *ServiceA* prints a string coming from *ServiceB*. The value of the string was “I’m from Site 1” on *ServiceB*₁ and “I’m from Site 2” on *ServiceB*₂. The request sent to *ServiceA*₁ was: `curl http://localhost:1234/ServiceA/e -H "scope: ServiceB/Site2"` and *ServiceA* printed “I’m from Site 2”.

As mentioned, this version required an endpoint for every service hard-coded in the Cheops core. Moreover, Consul and Envoy had much more functionalities we did not required, so we decided to do our own light service mesh.

This version of Cheops is available on the Cheops project page⁵.

5. <https://gitlab.inria.fr/discovery/cheops/-/tree/v0.1.0>

PUBLICATIONS



Conferences

ICSOC 2022 Delavergne, M., Antony, G.J., Lebre, A. (2022). Cheops, a Service to Blow Away Cloud Applications to the Edge. In: Troya, J., Medjahed, B., Piattini, M., Yao, L., Fernández, P., Ruiz-Cortés, A. (eds) Service-Oriented Computing. ICSOC 2022. Lecture Notes in Computer Science, vol 13740. Springer, Cham., doi: [10.1007/978-3-031-20984-0_37](https://doi.org/10.1007/978-3-031-20984-0_37).

Euro-Par 2021 Cherrueau, R.A., Delavergne, M., Lebre, A. (2021). Geo-distribute Cloud Applications at the Edge. In: Sousa, L., Roma, N., Tomás, P. (eds) Euro-Par 2021: Parallel Processing. Euro-Par 2021. Lecture Notes in Computer Science(), vol 12820. Springer, Cham., doi: [10.1007/978-3-030-85665-6_19](https://doi.org/10.1007/978-3-030-85665-6_19).

Journal

IEEE TPDS 2022 R.-A. Cherrueau et al., "EnosLib: A Library for Experiment-Driven Research in Distributed Computing", in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 6, pp. 1464-1477, 1 June 2022, doi: [10.1109/TPDS.2021.3111159](https://doi.org/10.1109/TPDS.2021.3111159).

Workshop

XP 2021 Workshops Delavergne, M., Cherrueau, R.A., Lebre, A. (2021). A Service Mesh for Collaboration Between Geo-Distributed Services: The Replication Case. In: Gregory, P., Kruchten, P. (eds) Agile Processes in Software Engineering and Extreme Programming – Workshops. XP 2021. Lecture Notes in Business Information Processing, vol 426. Springer, Cham. doi: [10.1007/978-3-030-88583-0_17](https://doi.org/10.1007/978-3-030-88583-0_17).



Research reports

RR 2022 Marie Delavergne, Geo Johns Antony, Adrien Lebre. Cheops, a service to blow away Cloud applications to the Edge. [Research Report] RR-9486, Inria Rennes - Bretagne Atlantique. 2022, pp.1-16. [⟨hal-03770492v2⟩](#).

RR 2020 Ronan-Alexandre Cherrueau, Marie Delavergne, Adrien Lebre, Javier Rojas Balderrama, Matthieu Simonin. Edge Computing Resource Management System: Two Years Later!. [Research Report] RR-9336, Inria Rennes Bretagne Atlantique. 2020. [⟨hal-02527366v2⟩](#).

Other content

OpenInfra Summit Geo Johns Antony, Marie Delavergne and Baptiste Jonglez. Cheops - Can a "service mesh" be the right solution for the Edge?, <https://www.youtube.com/watch?v=7EZ63DMRJhc>, OpenInfra Summit 2022, Berlin - Accessed: 2023-01-05.

PhD Defense slides Marie Delavergne - Cheops, a service mesh to geo-distribute (micro-)service applications at the Edge, <https://marie-donnie.github.io/assets/pdf/defense.pdf>

BIBLIOGRAPHY



- [1] Justin Blanchard, *The History of Cloud Computing*, <https://blog.servermania.com/the-history-of-cloud-computing/>, Accessed: 2022-08-22 (cit. on pp. 2, 148).
- [2] *What Is The Cloud - By AT&T*, https://ghostarchive.org/varchive/_a7hK6kWttE, Accessed: 2022-12-14 (cit. on pp. 2, 148).
- [3] Blesson Varghese, *History of the cloud*, <https://www.bcs.org/articles-opinion-and-research/history-of-the-cloud/>, Accessed: 2022-12-14 (cit. on pp. 2, 148).
- [4] Radoslav Ch., *37 Heavenly Cloud Computing Statistics for 2022*, <https://techjury.net/blog/cloud-computing-statistics/>, Accessed: 2022-09-05 (cit. on pp. 2, 148).
- [5] M. Satyanarayanan, « The Emergence of Edge Computing », in: *Computer* 50.1 (Jan. 2017), pp. 30–39, ISSN: 0018-9162 (cit. on pp. 2, 148).
- [6] Weisong Shi et al., « Edge Computing: Vision and Challenges », in: *IEEE Internet Things J.* 3.5 (2016), pp. 637–646, DOI: [10.1109/JIOT.2016.2579198](https://doi.org/10.1109/JIOT.2016.2579198), URL: <https://doi.org/10.1109/JIOT.2016.2579198> (cit. on pp. 2, 148).
- [7] David Espinel Sarmiento et al., « Multi-site Connectivity for Edge Infrastructures : DIMINET: DIstributed Module for Inter-site NETworking », in: *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, IEEE, 2020, pp. 121–130, DOI: [10.1109/CCGrid49817.2020.00-81](https://doi.org/10.1109/CCGrid49817.2020.00-81), URL: <https://doi.org/10.1109/CCGrid49817.2020.00-81> (cit. on pp. 2, 5, 49, 81, 148, 151).
- [8] *Open Source Cloud Computing Infrastructure - OpenStack*, <https://www.openstack.org/>, Accessed: 2022-10-20 (cit. on pp. 2, 148).
- [9] Athina Markopoulou et al., « Characterization of failures in an operational IP backbone network », in: *IEEE/ACM transactions on networking* 16.4 (2008), pp. 749–762 (cit. on pp. 2, 148).



- [10] Sabelo Dlamini et al., « Design of an autonomous management and orchestration for fog computing », in: *2018 International Conference on Intelligent and Innovative Computing Applications (ICONIC)*, IEEE, 2018, pp. 1–6 (cit. on pp. 2, 148).
- [11] Roger Pueyo Centelles et al., « A Monitoring System for Distributed Edge Infrastructures with Decentralized Coordination », in: *Algorithmic Aspects of Cloud Computing - 5th International Symposium, ALGO CLOUD 2019, Munich, Germany, September 10, 2019, Revised Selected Papers*, ed. by Ivona Brandic et al., vol. 12041, Lecture Notes in Computer Science, Springer, 2019, pp. 42–58, DOI: 10.1007/978-3-030-58628-7_4, URL: https://doi.org/10.1007/978-3-030-58628-7_4 (cit. on pp. 2, 148).
- [12] *The Discovery Initiative*, <https://beyondtheclouds.github.io/>, Accessed: 2022-11-09 (cit. on pp. 2, 5, 148, 151).
- [13] Weisong Shi et al., « Edge Computing: Vision and Challenges », in: *IEEE Internet Things J.* 3.5 (2016), pp. 637–646, DOI: 10.1109/JIOT.2016.2579198, URL: <https://doi.org/10.1109/JIOT.2016.2579198> (cit. on pp. 3, 149).
- [14] J. H. Saltzer, « Naming and binding of objects », in: *Operating Systems: An Advanced Course*, ed. by R. Bayer, R. M. Graham, and G. Seegmüller, Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 99–208, ISBN: 978-3-540-35880-0, DOI: 10.1007/3-540-08755-9_4, URL: https://doi.org/10.1007/3-540-08755-9_4 (cit. on pp. 3, 149).
- [15] Genc Tato et al., « Split and Migrate: Resource-Driven Placement and Discovery of Microservices at the Edge », in: *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, Neuchâtel, Switzerland*, vol. 153, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 9:1–9:16 (cit. on pp. 3, 20, 149).
- [16] Piergiorgio Bertoli et al., « Control Flow Requirements for Automated Service Composition », in: *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009*, IEEE Computer Society, 2009, pp. 17–24, DOI: 10.1109/ICWS.2009.31, URL: <https://doi.org/10.1109/ICWS.2009.31> (cit. on pp. 3, 149).



- [17] Adrien Lebre et al., « Revising OpenStack to Operate Fog/Edge Computing Infrastructures », *in: 2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, 2017, pp. 138–148, DOI: [10.1109/IC2E.2017.35](https://doi.org/10.1109/IC2E.2017.35), URL: <https://doi.org/10.1109/IC2E.2017.35> (cit. on pp. 3, 26, 149).
- [18] Dimitrios Vasilas, Marc Shapiro, and Bradley King, « A Modular Design for Geo-Distributed Querying », *in: CoRR abs/1803.04141* (2018), arXiv: [1803.04141](https://arxiv.org/abs/1803.04141), URL: <http://arxiv.org/abs/1803.04141> (cit. on pp. 3, 149).
- [19] Kun Ren, Dennis Li, and Daniel J. Abadi, « SLOG: Serializable, Low-latency, Geo-replicated Transactions », *in: Proc. VLDB Endow.* 12.11 (2019), pp. 1747–1761, DOI: [10.14778/3342263.3342647](https://doi.org/10.14778/3342263.3342647), URL: <http://www.vldb.org/pvldb/vol12/p1747-ren.pdf> (cit. on pp. 3, 149).
- [20] James C. Corbett et al., « Spanner: Google’s Globally-Distributed Database », *in: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 261–264 (cit. on pp. 3, 28, 149).
- [21] Marc Shapiro et al., « Just-Right Consistency: reconciling availability and safety », *in: CoRR abs/1801.06340* (2018), arXiv: [1801.06340](https://arxiv.org/abs/1801.06340), URL: <http://arxiv.org/abs/1801.06340> (cit. on pp. 3, 26, 149).
- [22] Marc Shapiro and Pierre Sutra, « Database Consistency Models », *in: Encyclopedia of Big Data Technologies*, 2019, DOI: [10.1007/978-3-319-63962-8_203-1](https://doi.org/10.1007/978-3-319-63962-8_203-1), URL: https://doi.org/10.1007/978-3-319-63962-8_203-1 (cit. on pp. 3, 149).
- [23] *CockroachDB: Distributed SQL - Cockroach Labs*, <https://www.cockroachlabs.com/product/>, Accessed: 2022-12-14 (cit. on pp. 3, 149).
- [24] R.-A. Cherrueau, *A POC of OpenStack Keystone over CockroachDB*, <https://beyondtheclouds.github.io/blog/openstack/cockroachdb/2017/12/22/a-poc-of-openstack-keystone-over-cockroachdb.html>, Accessed: 2022-09-25, Dec. 2017 (cit. on pp. 3, 26, 149).
- [25] Marie Delavergne, Ronan-Alexandre Cherrueau, and Adrien Lebre, *Evaluation of OpenStack Multi-Region Keystone Deployments*, <https://beyondtheclouds.github.io/blog/openstack/cockroachdb/2018/06/04/evaluation-of->



- [openstack-multi-region-keystone-deployments.html](#), Accessed: 2022-09-25, June 2018 (cit. on pp. 3, 5, 26, 111, 149, 151).
- [26] Synopsys, *Openstack code analysis*, <https://www.openhub.net/p/openstack>, Accessed: 2022-07-05 (cit. on pp. 3, 26, 149).
- [27] Nikolas Roman Herbst et al., « Elasticity in Cloud Computing: What It Is, and What It Is Not », in: *10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA: USENIX Association, June 2013, pp. 23–27, ISBN: 978-1-931971-02-7 (cit. on pp. 4, 150).
- [28] Wubin Li et al., « Service mesh: Challenges, state of the art, and future research opportunities », in: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, IEEE, 2019, pp. 122–1225 (cit. on pp. 4, 51, 52, 74, 77, 150).
- [29] Yu Gan et al., « An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems », in: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18 (cit. on pp. 4, 150).
- [30] William Morgan, *The Service Mesh*, <https://buoyant.io/service-mesh-manifesto>, Accessed: 2022-09-09 (cit. on pp. 4, 52, 150).
- [31] Ronan-Alexandre Cherrueau, Marie Delavergne, and Adrien Lèbre, « Geo-distribute Cloud Applications at the Edge », in: *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Proceedings*, ed. by Leonel Sousa, Nuno Roma, and Pedro Tomás, vol. 12820, Lecture Notes in Computer Science, Springer, 2021, pp. 301–316, DOI: [10.1007/978-3-030-85665-6_19](https://doi.org/10.1007/978-3-030-85665-6_19), URL: https://doi.org/10.1007/978-3-030-85665-6_19 (cit. on pp. 5, 16, 20, 26, 70, 71, 82, 151).
- [32] Marie Delavergne, Ronan-Alexandre Cherrueau, and Adrien Lebre, « A service mesh for collaboration between geo-distributed services: the replication case », in: *AMP 2021: 2nd International Workshop on Agility with Microservices Programming*, vol. 426, Lecture Notes in Business Information Processing (LNBIP) - Agile Processes in Software Engineering and Extreme Programming – XP 2021 Workshops, Online, France, June 2021, pp. 176–185, DOI: [10.1007/978-3-030-88583-0_17](https://doi.org/10.1007/978-3-030-88583-0_17), URL: <https://hal.inria.fr/hal-03282425> (cit. on pp. 5, 70, 151).



- [33] Geo Johns Antony, Marie Delavergne, and Baptiste Jonglez, *Cheops: a framework to geo-distribute micro-service applications at the Edge*, https://gitlab.inria.fr/discovery/cheops/-/raw/master/Infos/Slides_OpenInfra_Summit_2022.pdf, OpenInfra Summit 2022, Berlin - Accessed: 2022-10-06 (cit. on pp. 5, 70, 151).
- [34] Geo Johns Antony, Marie Delavergne, and Baptiste Jonglez, *Cheops - Can a "service mesh" be the right solution for the Edge?*, <https://www.youtube.com/watch?v=7EZ63DMRJhc>, OpenInfra Summit 2022, Berlin - Accessed: 2023-01-05 (cit. on pp. 5, 151).
- [35] Marie Delavergne, Geo Johns Antony, and Adrien Lebre, « Cheops, a Service to Blow Away Cloud Applications to the Edge », in: *Service-Oriented Computing*, ed. by Javier Troya et al., Cham: Springer Nature Switzerland, 2022, pp. 530–539, ISBN: 978-3-031-20984-0 (cit. on pp. 5, 85, 151).
- [36] Ronan-Alexandre Cherrueau et al., « Enoslib: A library for experiment-driven research in distributed computing », in: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (2021), pp. 1464–1477 (cit. on pp. 5, 151).
- [37] *EnOSlib: Surviving the homotogeneous world*, <https://discovery.gitlabpages.inria.fr/enoslib/>, Accessed: 2022-11-15 (cit. on pp. 5, 151).
- [38] *Juice*, <https://github.com/BeyondTheClouds/juice>, Accessed: 2022-11-15 (cit. on pp. 5, 151).
- [39] *Kubernetes*, <https://kubernetes.io/>, Accessed: 2022-10-20 (cit. on pp. 5, 151).
- [40] *12 Largest Data Centers In The World In 2022 [By Size]*, <https://www.rankred.com/largest-data-centers-in-the-world/>, Accessed: 2022-08-03 (cit. on p. 9).
- [41] Peter Mell, Tim Grance, et al., « The NIST definition of cloud computing », in: (2011) (cit. on p. 10).
- [42] Johannes Thönes, « Microservices », in: *IEEE Softw.* 32.1 (2015), p. 116, DOI: [10.1109/MS.2015.11](https://doi.org/10.1109/MS.2015.11), URL: <https://doi.org/10.1109/MS.2015.11> (cit. on p. 12).
- [43] Andy Davis, Jay Parikh, and William E Weihl, « Edgecomputing: extending enterprise applications to the edge of the internet », in: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 2004, pp. 180–187 (cit. on p. 15).



- [44] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun, « The Akamai Network: A Platform for High-Performance Internet Applications », *in: SIGOPS Oper. Syst. Rev.* 44.3 (2010), 2–19, ISSN: 0163-5980, DOI: [10.1145/1842733.1842736](https://doi.org/10.1145/1842733.1842736), URL: <https://doi.org/10.1145/1842733.1842736> (cit. on p. 15).
- [45] Milos Simic et al., « Towards Edge Computing as a Service: Dynamic Formation of the Micro Data-Centers », *in: IEEE Access* 9 (2021), pp. 114468–114484, DOI: [10.1109/ACCESS.2021.3104475](https://doi.org/10.1109/ACCESS.2021.3104475), URL: <https://doi.org/10.1109/ACCESS.2021.3104475> (cit. on p. 15).
- [46] Nan Wang et al., « ENORM: A framework for edge node resource management », *in: IEEE transactions on services computing* 13.6 (2017), pp. 1086–1099 (cit. on pp. 15, 21).
- [47] Jonathan McChesney et al., « Defog: fog computing benchmarks », *in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58 (cit. on p. 15).
- [48] Mahadev Satyanarayanan et al., « The Case for VM-Based Cloudlets in Mobile Computing », *in: IEEE Pervasive Computing* 8.4 (2009), pp. 14–23, DOI: [10.1109/MPRV.2009.82](https://doi.org/10.1109/MPRV.2009.82) (cit. on p. 15).
- [49] Enrique Saurez et al., « OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures », *in: SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, ed. by Carlo Curino, Georgia Koutrika, and Ravi Netravali, ACM, 2021, pp. 182–196, DOI: [10.1145/3472883.3487008](https://doi.org/10.1145/3472883.3487008), URL: <https://doi.org/10.1145/3472883.3487008> (cit. on pp. 15, 44, 50, 66, 67).
- [50] Ilyas Toumlilt, Pierre Sutra, and Marc Shapiro, « Highly-Available and Consistent Group Collaboration at the Edge with Colony », *in: Proceedings of the 22nd International Middleware Conference*, Middleware '21, Québec city, Canada: Association for Computing Machinery, 2021, 336–351, ISBN: 9781450385343, DOI: [10.1145/3464298.3493405](https://doi.org/10.1145/3464298.3493405), URL: <https://doi.org/10.1145/3464298.3493405> (cit. on pp. 15, 62–68).
- [51] Junjue Wang et al., « Towards scalable edge-native applications », *in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 152–165 (cit. on pp. 15, 58–61, 64, 66, 67).



- [52] *Global Ping Statistics*, <https://wondernetwork.com/pings/>, Accessed: 2022-10-20 (cit. on p. 16).
- [53] Wei Li et al., « On Enabling Sustainable Edge Computing with Renewable Energy Resources », in: *IEEE Communications Magazine* 56.5 (2018), pp. 94–101, DOI: [10.1109/MCOM.2018.1700888](https://doi.org/10.1109/MCOM.2018.1700888) (cit. on p. 17).
- [54] Blesson Varghese et al., « Revisiting the Arguments for Edge Computing Research », in: *IEEE Internet Comput.* 25.5 (2021), pp. 36–42, DOI: [10.1109/MIC.2021.3093924](https://doi.org/10.1109/MIC.2021.3093924), URL: <https://doi.org/10.1109/MIC.2021.3093924> (cit. on p. 17).
- [55] Hitoshi Ikezawa and Masakatsu Imafuku, « Convenience survey of IoT house equipment for a smart life », in: *2020 IEEE 2nd Global Conference on Life Sciences and Technologies (LifeTech)*, IEEE, 2020, pp. 290–294 (cit. on pp. 17, 18).
- [56] Vasuki Narasimha Swamy et al., « Low-Cost Aerial Imaging for Small Holder Farmers », in: *ACM Compass 2019*, ACM, 2019, URL: <https://www.microsoft.com/en-us/research/publication/low-cost-aerial-imaging-for-small-holder-farmers/> (cit. on p. 17).
- [57] Jian Ding and Ranveer Chandra, « Towards Low Cost Soil Sensing Using Wi-Fi », in: *MobiCom 2019*, ACM, 2019, URL: <https://www.microsoft.com/en-us/research/publication/towards-low-cost-soil-sensing-using-wi-fi/> (cit. on p. 17).
- [58] S. Jaiganesh, K. Gunaseelan, and V. Ellappan, « IOT agriculture to improve food and farming technology », in: *2017 Conference on Emerging Devices and Smart Systems (ICEDSS)*, 2017, pp. 260–266, DOI: [10.1109/ICEDSS.2017.8073690](https://doi.org/10.1109/ICEDSS.2017.8073690) (cit. on p. 17).
- [59] Safuriyawu Ahmed et al., « HyDiLLEch: a WSN-based Distributed Leak Detection and Localisation in Crude Oil Pipelines », in: *International Conference on Advanced Information Networking and Applications*, Springer, 2021, pp. 626–637 (cit. on p. 18).
- [60] Emmanuel N Aba et al., « Petroleum pipeline monitoring using an internet of things (IoT) platform », in: *SN Applied Sciences* 3.2 (2021), pp. 1–12 (cit. on p. 18).



- [61] Sheetal Vatari, Aarti Bakshi, and Tanvi Thakur, « Green house by using IOT and cloud computing », *in: 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, IEEE, 2016, pp. 246–250 (cit. on p. 18).
- [62] Wen-Tsai Sung and Sung-Jung Hsiao, « The application of thermal comfort control based on Smart House System of IoT », *in: Measurement* 149 (2020), p. 106997 (cit. on p. 18).
- [63] Bhagya Nathali Silva, Murad Khan, and Kijun Han, « Towards sustainable smart cities: A review of trends, architectures, components, and open challenges in smart cities », *in: Sustainable Cities and Society* 38 (2018), pp. 697–713 (cit. on p. 18).
- [64] ChuanTao Yin et al., « A literature survey on smart cities », *in: Science China Information Sciences* 58.10 (2015), pp. 1–18 (cit. on p. 18).
- [65] Antonio Miele et al., « A Runtime Resource Management and Provisioning Middleware for Fog Computing Infrastructures », *in: ACM Trans. Internet Things* 3.3 (2022), 17:1–17:29, DOI: [10.1145/3506718](https://doi.org/10.1145/3506718), URL: <https://doi.org/10.1145/3506718> (cit. on p. 18).
- [66] Blesson Varghese et al., « Challenges and opportunities in edge computing », *in: 2016 IEEE International Conference on Smart Cloud (SmartCloud)*, IEEE, 2016, pp. 20–26 (cit. on p. 18).
- [67] Platform9, *Fight Latency at the Edge with Kubernetes-Based Infrastructure – Part I*, <https://platform9.com/blog/fight-latency-at-the-edge-with-kubernetes-based-infrastructure-part-1/>, Accessed: 2022-09-23 (cit. on p. 18).
- [68] Weisong Shi et al., « Edge computing: Vision and challenges », *in: IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on pp. 18, 23).
- [69] Cheol-Ho Hong and Blesson Varghese, « Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms », *in: ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–37 (cit. on p. 18).
- [70] Yinhao Xiao et al., « Edge computing security: State of the art and challenges », *in: Proceedings of the IEEE* 107.8 (2019), pp. 1608–1631 (cit. on p. 18).



- [71] David Williams, *Edge Computing: Inherent Challenges and How to Address Them*, <https://www.rfcode.com/blog/edge-computing-inherent-challenges-and-how-to-address-them>, Accessed: 2022-09-25 (cit. on p. 18).
- [72] Shaun O'Meara, *Edge Computing Challenges*, <https://www.mirantis.com/blog/edge-computing-challenges/>, Accessed: 2022-09-25 (cit. on p. 18).
- [73] Shlomit Shaked and Tirza Routtenberg, « Identification of Edge Disconnections in Networks Based on Graph Filter Outputs », in: *IEEE Transactions on Signal and Information Processing over Networks* 7 (2021), pp. 578–594, DOI: [10.1109/TSIPN.2021.3107628](https://doi.org/10.1109/TSIPN.2021.3107628) (cit. on p. 18).
- [74] Marco Iorio et al., « Computing Without Borders: The Way Towards Liquid Computing », in: *arXiv preprint arXiv:2204.05710* (2022) (cit. on pp. 18, 40, 50, 51, 66, 67).
- [75] Mulugeta Ayalew Tamiru, « Automatic Resource Management in Geo-Distributed Multi-Cluster Environments. (Gestion automatique des ressources dans les environnements multi-clusters géo-distribués) », PhD thesis, University of Rennes 1, France, 2021, URL: <https://tel.archives-ouvertes.fr/tel-03351598> (cit. on p. 19).
- [76] Lara Lorna Jimenez and Olov Schelen, « HYDRA: Decentralized Location-aware Orchestration of Containerized Applications », in: *IEEE Transactions on Cloud Computing* (2020), pp. 1–1, DOI: [10.1109/TCC.2020.3041465](https://doi.org/10.1109/TCC.2020.3041465) (cit. on pp. 19, 41, 50, 66, 67, 109).
- [77] Maarten van Steen and Andrew S. Tanenbaum, *Distributed systems, 3rd Edition*, Maarten van Steen, 2017, ISBN: 978-1543057386 (cit. on p. 20).
- [78] *Local-first software*, <https://www.inkandswitch.com/local-first/>, Accessed: 2022-10-03 (cit. on p. 21).
- [79] Zhi Wang et al., « Cloud-based social application deployment using local processing and global distribution », in: *Conference on emerging Networking Experiments and Technologies, CoNEXT '12, Nice, France - December 10 - 13, 2012*, ed. by Chadi Barakat et al., ACM, 2012, pp. 301–312, DOI: [10.1145/2413176.2413211](https://doi.org/10.1145/2413176.2413211), URL: <https://doi.org/10.1145/2413176.2413211> (cit. on p. 21).
- [80] Pooyan Jamshidi et al., « Microservices: The Journey So Far and Challenges Ahead », in: *IEEE Softw.* 35.3 (2018), pp. 24–35 (cit. on p. 21).



- [81] Roy Thomas Fielding, « Architectural Styles and the Design of Network-based Software Architectures », AAI9980887, PhD thesis, University of California, Irvine, 2000, ISBN: 0-599-87118-0 (cit. on pp. 21, 73).
- [82] Mathew Ryden et al., « Nebula: Distributed edge cloud for data intensive computing », in: *2014 IEEE International Conference on Cloud Engineering*, IEEE, 2014, pp. 57–66 (cit. on p. 21).
- [83] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu, « Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds », in: *IEEE Transactions on Cloud Computing* 7.2 (2016), pp. 329–343 (cit. on p. 21).
- [84] H. M. N. Dilum Bandara and Anura P. Jayasumana, « Collaborative applications over peer-to-peer systems-challenges and solutions », in: *Peer Peer Netw. Appl.* 6.3 (2013), pp. 257–276, DOI: [10.1007/s12083-012-0157-3](https://doi.org/10.1007/s12083-012-0157-3), URL: <https://doi.org/10.1007/s12083-012-0157-3> (cit. on p. 21).
- [85] Rajkumar Buyya, Toni Cortes, and Hai Jin, « Single system image », in: *The International Journal of High Performance Computing Applications* 15.2 (2001), pp. 124–135 (cit. on p. 21).
- [86] Ronan-Alexandre Cherrueau, Adrien Lebre, and Pierre Riteau, *Toward Fog, Edge, and NFV Deployments: Evaluating OpenStack WANwide*, OpenStack Summit, Boston (MA) USA, <https://www.youtube.com/watch?v=xwT08H02Nok>, Accessed: 11/2022, May 2017 (cit. on p. 22).
- [87] Zhenyu Fan et al., « Serving at the Edge: An Edge Computing Service Architecture Based on ICN », in: *ACM Trans. Internet Techn.* 22.1 (2022), 22:1–22:27, DOI: [10.1145/3464428](https://doi.org/10.1145/3464428), URL: <https://doi.org/10.1145/3464428> (cit. on p. 22).
- [88] Schahram Dustdar and Wolfgang Schreiner, « A survey on web services composition », in: *IJWGS* 1.1 (2005), pp. 1–30, URL: <http://dx.doi.org/10.1504/IJWGS.2005.007545> (cit. on p. 22).
- [89] Dipanjan Chakraborty and Anupam Joshi, *Dynamic Service Composition: State-of-the-Art and Research Directions*, tech. rep., 2001 (cit. on p. 22).
- [90] Keita Fujii and Tatsuya Suda, « Dynamic service composition using semantic information », in: *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*, 2004,



- pp. 39–48, DOI: [10.1145/1035167.1035174](https://doi.org/10.1145/1035167.1035174), URL: <https://doi.org/10.1145/1035167.1035174> (cit. on p. 22).
- [91] Mulugeta Ayalew Tamiru et al., « mck8s: An orchestration platform for geo-distributed multi-cluster environments », in: *30th International Conference on Computer Communications and Networks, ICCCN 2021, Athens, Greece, July 19-22, 2021*, IEEE, 2021, pp. 1–10, DOI: [10.1109/ICCCN52240.2021.9522318](https://doi.org/10.1109/ICCCN52240.2021.9522318), URL: <https://doi.org/10.1109/ICCCN52240.2021.9522318> (cit. on pp. 22, 37).
- [92] Agostino Forestiero et al., « A proximity-based self-organizing framework for service composition and discovery », in: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, IEEE, 2010, pp. 428–437 (cit. on p. 23).
- [93] Jongbeom Lim, « Scalable Fog Computing Orchestration for Reliable Cloud Task Scheduling », in: *Applied Sciences* 11.22 (2021), p. 10996 (cit. on p. 23).
- [94] Rüdiger Schollmeier, « A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications », in: *1st International Conference on Peer-to-Peer Computing (P2P 2001), 27-29 August 2001, Linköping, Sweden*, ed. by Ross Lee Graham and Nahid Shahmehri, IEEE Computer Society, 2001, pp. 101–102, DOI: [10.1109/P2P.2001.990434](https://doi.org/10.1109/P2P.2001.990434), URL: <https://doi.org/10.1109/P2P.2001.990434> (cit. on p. 23).
- [95] Ronan-Alexandre Cherrueau et al., « Edge Computing Resource Management System: a Critical Building Block! Initiating the debate via OpenStack », in: *USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2018, Boston, MA, July 10*. USENIX Association, 2018 (cit. on pp. 23, 27).
- [96] Ronan-Alexandre Cherrueau et al., *Edge Computing Resource Management System: Two Years Later!*, Research Report RR-9336, Inria Rennes Bretagne Atlantique, Apr. 2020, URL: <https://hal.inria.fr/hal-02527366> (cit. on pp. 24, 26, 70, 90, 111).
- [97] Peter Alvaro et al., « Consistency Analysis in Bloom: a CALM and Collected Approach », in: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 249–260 (cit. on p. 24).



- [98] Andrew P. Black et al., « Distribution and Abstract Types in Emerald », *in: IEEE Trans. Software Eng.* 13.1 (1987), pp. 65–76, DOI: [10.1109/TSE.1987.232836](https://doi.org/10.1109/TSE.1987.232836), URL: <https://doi.org/10.1109/TSE.1987.232836> (cit. on p. 24).
- [99] Robert H. Halstead Jr., « MULTILISP: A Language for Concurrent Symbolic Computation », *in: ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538, ISSN: 0164-0925, DOI: [10.1145/4472.4478](https://doi.org/10.1145/4472.4478), URL: <http://doi.acm.org/10.1145/4472.4478> (cit. on p. 24).
- [100] Daniel Abadi, « Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story », *in: Computer* 45.2 (2012), pp. 37–42 (cit. on p. 26).
- [101] Ayoub Bousselmi, Jean-François Peltier, and Abdelhadi Chari, « Towards a massively distributed IaaS operating system: Composition and evaluation of OpenStack », *in: 2016 IEEE Conference on Standards for Communications and Networking, CSCN 2016, Berlin, Germany, October 31 - November 2, 2016*, 2016, pp. 288–293, DOI: [10.1109/CSCN.2016.7785190](https://doi.org/10.1109/CSCN.2016.7785190), URL: <https://doi.org/10.1109/CSCN.2016.7785190> (cit. on p. 26).
- [102] Open Infrastructure Foundation, *Introduction to Keystone Federation*, <https://docs.openstack.org/keystone/latest/admin/federation/introduction.html>, Accessed: 2022-07-05 (cit. on p. 27).
- [103] *Image handling in edge environment*, https://wiki.openstack.org/wiki/Image_handling_in_edge_environment, Accessed: 2022-09-25 (cit. on pp. 27, 28).
- [104] Florian Daniel and Barbara Pernici, « Insights into Web Service Orchestration and Choreography », *in: Int. J. E Bus. Res.* 2.1 (2006), pp. 58–77, DOI: [10.4018/jebr.2006010104](https://doi.org/10.4018/jebr.2006010104), URL: <https://doi.org/10.4018/jebr.2006010104> (cit. on p. 36).
- [105] Breno G. S. Costa et al., « Orchestration in Fog Computing: A Comprehensive Survey », *in: ACM Comput. Surv.* 55.2 (2023), 29:1–29:34, DOI: [10.1145/3486221](https://doi.org/10.1145/3486221), URL: <https://doi.org/10.1145/3486221> (cit. on p. 36).
- [106] Oana-Mihaela Ungureanu, Calin Vladeanu, and Robert E. Kooij, « Collaborative Cloud - Edge: A Declarative API orchestration model for the NextGen 5G Core », *in: 15th IEEE International Conference on Service-Oriented System Engineering, SOSE 2021, Oxford, United Kingdom, August 23-26, 2021*, IEEE, 2021, pp. 124–



- 133, DOI: [10.1109/SOSE52839.2021.00019](https://doi.org/10.1109/SOSE52839.2021.00019), URL: <https://doi.org/10.1109/SOSE52839.2021.00019> (cit. on p. 37).
- [107] Amjad Ullah et al., « MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum », in: *J. Grid Comput.* 19.4 (2021), p. 47, DOI: [10.1007/s10723-021-09589-5](https://doi.org/10.1007/s10723-021-09589-5), URL: <https://doi.org/10.1007/s10723-021-09589-5> (cit. on p. 37).
- [108] Cecil Wöbker et al., « Fogernetes: Deployment and management of fog computing applications », in: *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2018, pp. 1–7 (cit. on p. 37).
- [109] Alina Buzachis et al., « Towards Osmotic Computing: a Blue-Green Strategy for the Fast Re-Deployment of Microservices », in: *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–6, DOI: [10.1109/ISCC47284.2019.8969621](https://doi.org/10.1109/ISCC47284.2019.8969621) (cit. on p. 37).
- [110] Daniele Santoro et al., « Foggy: A Platform for Workload Orchestration in a Fog Computing Environment », in: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 231–234, DOI: [10.1109/CloudCom.2017.62](https://doi.org/10.1109/CloudCom.2017.62) (cit. on p. 37).
- [111] Tom Goethals, Filip De Turck, and Bruno Volckaert, « FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices », in: *Internet of Vehicles. Technologies and Services Toward Smart Cities - 6th International Conference, IOV 2019, Kaohsiung, Taiwan, November 18-21, 2019, Proceedings*, ed. by Ching-Hsien Hsu et al., vol. 11894, Lecture Notes in Computer Science, Springer, 2019, pp. 174–189, DOI: [10.1007/978-3-030-38651-1_16](https://doi.org/10.1007/978-3-030-38651-1_16), URL: https://doi.org/10.1007/978-3-030-38651-1_16 (cit. on p. 37).
- [112] Sebastian Böhm and Guido Wirtz, « Towards orchestration of cloud-edge architectures with kubernetes », in: *International Summit Smart City 360°*, Springer, 2022, pp. 207–230 (cit. on p. 37).
- [113] Karim Manaouil and Adrien Lebre, *Kubernetes and the Edge?*, Research Report RR-9370, Inria Rennes - Bretagne Atlantique, Oct. 2020, p. 19, URL: <https://hal.inria.fr/hal-02972686> (cit. on pp. 37, 67).
- [114] *Kubernetes Cluster Federation*, <https://github.com/kubernetes-sigs/kubefed>, Accessed: 2022-09-23 (cit. on p. 37).



- [115] *Submariner*, <https://github.com/submariner-io/submariner>, Accessed: 2022-09-23 (cit. on p. 37).
- [116] Ying Xiong et al., « Extend Cloud to Edge with KubeEdge », in: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377, DOI: [10.1109/SEC.2018.00048](https://doi.org/10.1109/SEC.2018.00048) (cit. on p. 37).
- [117] *KubeEdge, a Kubernetes Native Edge Computing Framework*, <https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro>, Accessed: 10/2022 (cit. on p. 37).
- [118] *StarlingX, a complete cloud infrastructure software stack for the edge*, <https://www.starlingx.io>, Accessed: 11/2022 (cit. on p. 37).
- [119] Adrian F. Spataru, « Decentralized and Fault Tolerant Cloud Service Orchestration », in: *Scalable Comput. Pract. Exp.* 21.4 (2020), pp. 709–725, DOI: [10.12694/scpe.v21i4.1838](https://doi.org/10.12694/scpe.v21i4.1838), URL: <https://doi.org/10.12694/scpe.v21i4.1838> (cit. on pp. 37, 50, 51, 66, 67).
- [120] Andreas Tsagkaropoulos et al., « Extending TOSCA for Edge and Fog Deployment Support », in: *Electronics* 10.6 (2021), ISSN: 2079-9292, DOI: [10.3390/electronics10060737](https://doi.org/10.3390/electronics10060737), URL: <https://www.mdpi.com/2079-9292/10/6/737> (cit. on p. 38).
- [121] Cloud Native Computing Foundation, *OpenYurt*, <https://openyurt.io/>, Accessed: 2022-09-09 (cit. on pp. 45, 50, 51, 66, 67).
- [122] João Monteiro Soares et al., « Re-designing Cloud Platforms for Massive Scale Using a P2P Architecture », in: *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*, IEEE Computer Society, 2017, pp. 57–64, DOI: [10.1109/CloudCom.2017.17](https://doi.org/10.1109/CloudCom.2017.17), URL: <https://doi.org/10.1109/CloudCom.2017.17> (cit. on pp. 47, 50, 51, 66, 67).
- [123] Xin Han, « Scaling OpenStack clouds using peer-to-peer technologies », MA thesis, 2017 (cit. on p. 47).
- [124] OpenInfra Foundation, *Tricircle*, <https://wiki.openstack.org/wiki/Tricircle>, Accessed: 2022-09-21 (cit. on p. 48).
- [125] *What's a service mesh?*, <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh>, Accessed: 2022-10-01 (cit. on p. 51).



- [126] *What is a service mesh?*, <https://linkerd.io/what-is-a-service-mesh/>, Accessed: 2022-09-09 (cit. on p. 51).
- [127] *Reverse proxy*, https://en.wikipedia.org/wiki/Reverse_proxy, Accessed: 10/2022 (cit. on p. 52).
- [128] Mrittika Ganguli et al., « Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge », in: *2021 IEEE International Conference on Edge Computing (EDGE)*, IEEE, 2021, pp. 78–85 (cit. on pp. 52, 57).
- [129] Rahul Sharma and Avinash Singh, « Introduction to the Service Mesh », in: *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes*, Berkeley, CA: Apress, 2020, pp. 47–98, ISBN: 978-1-4842-5458-5, DOI: [10.1007/978-1-4842-5458-5_2](https://doi.org/10.1007/978-1-4842-5458-5_2), URL: https://doi.org/10.1007/978-1-4842-5458-5_2 (cit. on pp. 53, 66–68).
- [130] *Linkerd*, <https://linkerd.io/>, Accessed: 2022-09-23 (cit. on pp. 55, 66, 67).
- [131] Jinke Ren et al., « Collaborative Cloud and Edge Computing for Latency Minimization », in: *IEEE Trans. Veh. Technol.* 68.5 (2019), pp. 5031–5044, DOI: [10.1109/TVT.2019.2904244](https://doi.org/10.1109/TVT.2019.2904244), URL: <https://doi.org/10.1109/TVT.2019.2904244> (cit. on p. 58).
- [132] Kiryong Ha et al., « Towards Wearable Cognitive Assistance », in: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, Bretton Woods, New Hampshire, USA: Association for Computing Machinery, 2014, 68–81, ISBN: 9781450327930, DOI: [10.1145/2594368.2594383](https://doi.org/10.1145/2594368.2594383), URL: <https://doi.org/10.1145/2594368.2594383> (cit. on p. 58).
- [133] Ilyas Toumlilt, « Colony : a Hybrid Consistency System for Highly-Available Collaborative Edge Computing », Theses, Sorbonne Université, Dec. 2021, URL: <https://tel.archives-ouvertes.fr/tel-03727724> (cit. on p. 62).
- [134] Mustaque Ahamad et al., « Causal memory: Definitions, implementation, and programming », in: *Distributed Computing 9.1* (1995), pp. 37–49 (cit. on p. 63).
- [135] Marek Zawirski et al., « Write fast, read in the past: Causal consistency for client-side applications », in: *Proceedings of the 16th Annual Middleware Conference*, 2015, pp. 75–87 (cit. on p. 63).



- [136] Ronan-Alexandre "Cherrueau, Adrien Lebre, and Javier" Rojas Balderrama, *Implementing Localization into OpenStack CLI for a Free Collaboration of Edge OpenStack Clouds*, OpenStack Summit, Denver, USA, <https://www.openstack.org/videos/summits/denver-2019/implementing-localization-into-openstack-cli-for-a-free-collaboration-of-edge-openstack-clouds>, Dec. 2019 (cit. on pp. 70, 90, 109).
- [137] Geo Johns Antony and Marie Delavergne, *Cheops: a framework to geo-distribute micro-service applications at the Edge*, https://gitlab.inria.fr/discovery/cheops/-/raw/master/Infos/Poster_Compas_2022.pdf, Poster for Conférence francophone d'informatique en Parallélisme, Architecture et Système - Accessed: 2022-10-06 (cit. on p. 70).
- [138] Marie Delavergne, Geo Johns Antony, and Adrien Lebre, *Cheops, a service to blow away Cloud applications to the Edge*, Research Report RR-9486, Inria Rennes - Bretagne Atlantique, Sept. 2022, pp. 1–16, URL: <https://hal.inria.fr/hal-03770492> (cit. on pp. 70, 85).
- [139] Florian Rosenberg et al., « Composing restful services and collaborative workflows: A lightweight approach », in: *IEEE Internet computing 12.5* (2008), pp. 24–31 (cit. on p. 71).
- [140] Rosa Alarcón, Erik Wilde, and Jesus Bellido, « Hypermedia-Driven RESTful Service Composition », in: *Service-Oriented Computing - ICSOC 2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG, San Francisco, CA, USA, December 7-10, 2010, Revised Selected Papers*, ed. by E. Michael Maximilien et al., vol. 6568, Lecture Notes in Computer Science, 2010, pp. 111–120, DOI: [10.1007/978-3-642-19394-1_12](https://doi.org/10.1007/978-3-642-19394-1_12), URL: https://doi.org/10.1007/978-3-642-19394-1_12 (cit. on p. 71).
- [141] Florian Haupt et al., « Service composition for REST », in: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, IEEE, 2014, pp. 110–119 (cit. on p. 71).
- [142] Barbara Liskov, « A design methodology for reliable software systems », in: *American Federation of Information Processing Societies: Proceedings of the AFIPS '72 Fall Joint Computer Conference, December 5-7, USA, 1972*, pp. 191–199 (cit. on p. 73).



- [143] David Lorge Parnas, « On the Criteria To Be Used in Decomposing Systems into Modules », in: *Commun. ACM* 15.12 (1972), pp. 1053–1058 (cit. on p. 73).
- [144] influxdata, *InfluxDB*, <https://www.influxdata.com/>, Accessed: 2022-07-12 (cit. on p. 75).
- [145] Cheng Li et al., « Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary », in: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, ed. by Chandu Thekkath and Amin Vahdat, USENIX Association, 2012, pp. 265–278, URL: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li> (cit. on p. 83).
- [146] Ronan-Alexandre Cherrueau, Javier Rojas Balderrama, and Matthieu Simonin, *OpenStackoïd: Make your OpenStacks Collaborative*, <https://gitlab.inria.fr/discovery/openstackoid>, Accessed: 11/2022 (cit. on p. 90).
- [147] Deepthi Devaki Akkoorath et al., « Cure: Strong semantics meets high availability and low latency », in: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2016 (cit. on p. 96).
- [148] Y. Zhu and Y. Wang, « SHAFT: Supporting Transactions with Serializability and Fault-Tolerance in Highly-Available Datastores », in: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 717–724 (cit. on p. 96).
- [149] Diego Ongaro and John Ousterhout, « In search of an understandable consensus algorithm », in: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014 (cit. on p. 100).
- [150] Daniel Balouek et al., « Adding Virtualization Capabilities to the Grid’5000 Testbed », in: *Cloud Computing and Services Science*, ed. by Ivan I. Ivanov et al., vol. 367, Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4, DOI: [10.1007/978-3-319-04519-1_1](https://doi.org/10.1007/978-3-319-04519-1_1) (cit. on pp. 100, 123).
- [151] *What Are API Gateways?*, <https://www.ibm.com/cloud/blog/api-gateway>, Accessed: 2022-10-27 (cit. on p. 107).



- [152] João Monteiro Soares et al., « Re-designing Cloud Platforms for Massive Scale Using a P2P Architecture », in: *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*, IEEE Computer Society, 2017, pp. 57–64, DOI: [10.1109/CloudCom.2017.17](https://doi.org/10.1109/CloudCom.2017.17), URL: <https://doi.org/10.1109/CloudCom.2017.17> (cit. on p. 108).
- [153] Iulian Moraru, David G Andersen, and Michael Kaminsky, « There is more consensus in egalitarian parliaments », in: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372 (cit. on p. 109).
- [154] Sarah Tollman, Seo Jin Park, and John Ousterhout, « {EPaxos} Revisited », in: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 613–632 (cit. on p. 109).
- [155] Nuno Preguiça, Carlos Baquero, and Marc Shapiro, « Conflict-Free Replicated Data Types (CRDTs) », in: *Encyclopedia of Big Data Technologies*, Springer, May 2018, DOI: [10.1007/978-3-319-63962-8_185-1](https://doi.org/10.1007/978-3-319-63962-8_185-1), eprint: [1805.06358](https://doi.org/10.1007/978-3-319-63962-8_185-1) (cit. on p. 109).
- [156] Marc Shapiro et al., « Conflict-free Replicated Data Types », in: *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, Springer LNCS volume 6976, Oct. 2011, pp. 386–400, DOI: [10.1007/978-3-642-24550-3_29](https://doi.org/10.1007/978-3-642-24550-3_29) (cit. on p. 109).
- [157] David G. Clarke, John M. Potter, and James Noble, « Ownership Types for Flexible Alias Protection », in: *SIGPLAN Not.* 33.10 (Oct. 1998), 48–64, ISSN: 0362-1340, DOI: [10.1145/286942.286947](https://doi.org/10.1145/286942.286947) (cit. on p. 114).
- [158] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira, « Ownership Types for Object Encapsulation », in: *SIGPLAN Not.* 38.1 (Jan. 2003), 213–223, ISSN: 0362-1340, DOI: [10.1145/640128.604156](https://doi.org/10.1145/640128.604156) (cit. on p. 114).
- [159] Emre Yigitoglu et al., « Distributed Orchestration in Large-Scale IoT Systems », in: *IEEE International Congress on Internet of Things, ICIOT 2017, Honolulu, HI, USA, June 25-30, 2017*, IEEE Computer Society, 2017, pp. 58–65, DOI: [10.1109/IEEE.ICIOT.2017.16](https://doi.org/10.1109/IEEE.ICIOT.2017.16), URL: <https://doi.org/10.1109/IEEE.ICIOT.2017.16>.
- [160] Fabienne Boyer et al., « A declarative approach for updating distributed microservices », in: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018, pp. 392–393.



- [161] Sara Bouchenak et al., « From Autonomic to Self-Self Behaviors: The JADE Experience », in: *ACM Trans. Auton. Adapt. Syst.* 6.4 (2011), 28:1–28:22, DOI: [10.1145/2019591.2019597](https://doi.org/10.1145/2019591.2019597), URL: <https://doi.org/10.1145/2019591.2019597>.
- [162] *What is Cluster Peering?*, <https://developer.hashicorp.com/consul/docs/connect/cluster-peering>, Accessed: 2022-10-17.
- [163] Luciano Baresi, Danilo Filgueira Mendonça, and Giovanni Quattrocchi, « PAPS: A Framework for Decentralized Self-management at the Edge », in: *Service-Oriented Computing - 17th International Conference, ICSOC 2019, Toulouse, France, October 28-31, 2019, Proceedings*, ed. by Sami Yangui et al., vol. 11895, Lecture Notes in Computer Science, Springer, 2019, pp. 508–522, DOI: [10.1007/978-3-030-33702-5_39](https://doi.org/10.1007/978-3-030-33702-5_39), URL: https://doi.org/10.1007/978-3-030-33702-5_39.
- [164] Alexander Bergmayr et al., « A Systematic Review of Cloud Modeling Languages », in: *ACM Comput. Surv.* 51.1 (2018), 22:1–22:38, DOI: [10.1145/3150227](https://doi.org/10.1145/3150227), URL: <https://doi.org/10.1145/3150227>.
- [165] Nabor C. Mendonça et al., « Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices », in: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA 2018, Madrid, Spain, September 24-28, 2018*, ed. by Jennifer Pérez, Raffaella Mirandola, and Hong-Mei Chen, ACM, 2018, 18:1–18:6, DOI: [10.1145/3241403.3241423](https://doi.org/10.1145/3241403.3241423), URL: <https://doi.org/10.1145/3241403.3241423>.

Résumé





Depuis l'utilisation du terme *Cloud* (nuage) dans ce contexte dans les années 1990⁶ [2, 3], le paradigme de l'informatique en nuage, ou nuagique (Cloud Computing, ou en plus court, Cloud) est devenu un pilier des mécanismes informatiques, en offrant des solutions pour les entreprises, les scientifiques, les particuliers. Il est désormais omniprésent, et beaucoup d'entreprises (plus de 60% [4]) utilisent des espaces de travail dans le nuage, qu'ils soient privés ou publics.

Traditionnellement, à l'exception du Cloud privé, d'énormes centres de données (datacenters, abrégés DCs) sont construits dans des endroits stratégiques (par exemple, en terme de coût énergétique) pour répondre aux demandes des utilisateurs du monde entier, ou du moins de larges parties du globe.

Cependant, il existe un besoin important pour les applications sensibles à la latence (pour lesquelles la latence doit être faible) d'être exécutées au plus près des *clients* (clients dans le sens de consommateurs de l'application, qui ne sont pas forcément des utilisateurs humains, mais qui peuvent être des applications liées à l'Internet des objets (IoT), aux villes intelligentes, etc.). Ce besoin est rempli par le nouveau paradigme d'informatique en périphérie, ou périphérie (Edge Computing) [5]. Son principe central est d'avoir de multiples micro et nano DCs à la périphérie du réseau, plus proches des clients [6]. Par exemple, les points de présence (PoP) à la périphérie du réseau pourraient être exploités pour obtenir cette infrastructure géo-distribuée (géographiquement distribuée sur le globe entier), à proximité des clients [7].

Initialement, les activités que j'ai menées sur le sujet se sont concentrées sur la révision d'un système de gestion des ressources tel qu'OpenStack [8] pour gérer ces infrastructures spécifiques à l'Edge Computing (ou Edge). Pour bénéficier de la géo-distribution de ces infrastructures, les systèmes distribués en périphérie doivent faire face à des latences élevées (entre des sites très éloignés les uns des autres) et de fréquentes déconnexions inhérentes aux réseaux étendus (Wide Area Network, abrégé WAN) [5, 9].

Afin de gérer une application sur une infrastructure aussi largement géo-distribuée, il convient également de tenir compte de son extensibilité, de la localisation des ressources et de sa résilience aux déconnexions ou autres défaillances des sites.

Pour faire face à ces défis, une direction possible serait de construire des applications spécifiquement pour l'Edge, en gardant à l'esprit ces difficultés [10, 11]. L'initiative Discovery [12] a suivi différentes directions ; lorsque j'ai commencé à travailler dans l'équipe, l'idée principale était de gérer les infrastructures en périphérie en révisant une plateforme

6. l'invention du Cloud en lui-même était bien plus ancien, autour des années 1950s/1960s [1].



de gestion d'infrastructure en nuage (à savoir OpenStack), pour la faire fonctionner à la périphérie.

L'approche initiale consistait à conserver la majeure partie de l'application sur le Cloud, où sont alors traitées les demandes d'opérations non critiques, et de traiter les autres requêtes (sensibles à la latence) sur des nœuds plus petits, mais plus nombreux et hétérogènes déployés à la périphérie [13]. En d'autres termes, des instances de chaque service critique d'une application doivent être déployées sur les sites de la périphérie pour atteindre les objectifs liés au paradigme (principalement de latence). Ce déploiement d'instances multiples est un problème en raison de la conservation des états de certains services (stateful) [14, 15], notamment car dans la plupart des scénarios de la vie réelle, les services conservent l'état des ressources qu'ils gèrent [16]. Pour clarifier, diviser un service qui ne conserve pas d'état (stateless) est simple, et il convient principalement diviser une application en ses différents microservices et décider de ce qui peut être distribué et ce qui doit rester centralisé [15]. Mais dans le cas de services à états, le problème est bien plus complexe, car il faut traiter les problèmes de synchronisation d'une manière spécifique pour chaque service de chaque application en ayant donc besoin de connaître le fonctionnement intégral de chaque service.

Pour aider à résoudre cette énigme, l'une des directions de recherche que nous avons suivie est de s'intéresser au partage des ressources entre les différents services directement au niveau de la base de données [17, 18, 19]. Cette solution consiste à utiliser une base de données distribuée de façon globale, comme un espace mémoire partagé que les services peuvent utiliser [20]. L'hypothèse sous-jacente est que les développeurs peuvent alors écrire une application au-dessus de ces services sans se soucier de la distribution/localité [21, 22]. Cette approche n'est cependant pas réellement simple car elle nécessite d'étudier la manière d'introduire la géo-distribution dans le code de l'application pour gérer les ressources sur une telle base de données, ce qui semble contradictoire avec l'hypothèse de base. Cette contradiction est plus largement expliquée dans la sous section 3.3.1 de la partie I, mais globalement, c'est une question de contexte d'exécution qui conduit à un code dédié. Nous avons découvert ceci en utilisant une base de données distribuée, à savoir CockroachDB [23], pour géo-distribuer OpenStack [24, 25]. OpenStack est un système énorme, comprenant environ 13 millions de lignes de code [26] et est utilisé principalement pour le Cloud. Par conséquent, la modification du code pour pouvoir gérer la géo-distribution des ressources n'est pas souhaitée car certaines applications natives du Cloud peuvent être énormes et ce serait donc un travail titanesque.



De plus, gérer la localisation des ressources demande non seulement des efforts considérables, mais aussi d'intégrer les aspects de la géo-distribution (partage de l'état entre les services sur plusieurs sites) dans le code métier, ce qui va à l'encontre du principe de séparation des préoccupations. Ce principe de séparation des préoccupations est largement adopté dans l'informatique en nuage où il existe un cloisonnement strict entre les équipes de développement et d'exploitation (abrégé DevOps) [27, 28]: Les programmeurs se concentrent sur le développement et la logique métier de l'application (c'est-à-dire les services), tandis que les DevOps sont en charge de l'exécution de l'application sur l'infrastructure (par exemple, le déploiement, la surveillance, la mise à l'échelle).

Un principe qui est mis en œuvre dans le monde du Cloud computing grâce à un concept appelé "*service mesh*" (littéralement "maillage de services"). Un service mesh repose sur le fait qu'une application dans l'informatique en nuage est représentée comme un ensemble de services faiblement couplés [29] pour atténuer la complexité opérationnelle associée aux applications modernes [28], ce qui permet de bien la séparer du code métier de l'application [30]. Ainsi, les service meshes sont la solution que j'ai étudiée pour maintenir les préoccupations de géo-distribution en dehors du code de l'application.

Ces problèmes ont motivé le travail que je défends dans ce manuscrit. Je propose une nouvelle approche qui s'appuie sur la modularité des applications du Cloud existantes, fonctionnant par services, pour exploiter les infrastructures géo-distribuées. Cette approche est générique à toutes les applications qui correspondent à ces règles :

Basée sur les services L'application doit être modulaire et avoir différents services gérant différentes ressources.

RESTful Les services de l'application doivent être conformes à la logique REST lorsqu'ils communiquent entre eux.

Thèmes et questions de recherche

En suivant le raisonnement présenté, cette thèse vise à trouver une manière générique d'utiliser les applications du nuage en périphérie, avec un impact minimal, voire nul sur leur code original.

Concrètement, les questions de recherche que nous abordons sont les suivantes :

- **Est-il possible d'utiliser des applications développées pour l'informatique en nuage sur des infrastructures en périphérie sans modifier leur code ?**



- Plus spécifiquement, une telle une telle approche peut-elle être utilisée pour gérer une infrastructure en périphérie, géo-distribuée, avec une application conçue de gestion du nuage ?
- Et en particulier, puisque les service meshes sont conçus pour gérer les communications entre les services d’une application en dehors de son code métier, peuvent-ils être une solution pour utiliser des applications du nuage sur des infrastructures en périphérie sans changer leur code ?

Contributions

Mon travail dans cette thèse a apporté trois contributions distinctes :

- La première contribution concerne la théorie de l’approche, qui a été qui a été présentée à Euro-Par 2021 [31], et une contribution plus spécifique sur la réplication présentée à l’atelier XP 2021 Workshops [32].
- La seconde contribution est la mise en œuvre de l’approche par une preuve de concept (Proof of Concept) appelée Cheops, et présentée à l’Open Infrastructure Summit [33, 34] et sous la forme d’un article court à l’ICSOC 2022 [35].
- Pour valider ce prototype, ainsi que les études préliminaires, j’ai également contribué à la proposition Enoslib [36, 37], une bibliothèque destinée à faciliter les expérimentations sur différentes infrastructures, que nous n’abordons pas dans ce manuscrit, car elle est hors sujet.

Pour donner un aperçu de mon travail pendant ma thèse, bien qu’elle ait été financée par Inria, j’ai également travaillé avec des personnes d’Orange Labs à travers différents projets de l’initiative Discovery [12], tels que [36, 25, 38, 7], sur lesquels ma thèse est basée.

J’ai commencé à travailler dans l’équipe sur OpenStack, puis le travail a été élargi pour inclure Kubernetes [39], avec des perspectives orientées pour tester plus d’applications afin d’assurer la généricité de la solution. De plus, cette approche fonctionnant à la fois sur OpenStack et Kubernetes permet de répondre en partie à la deuxième question de recherche, puisque ces deux systèmes permettent de gérer des applications sur les infrastructures nuagiques. Comprendre ces énormes applications pour savoir comment les manipuler a fait donc également partie de mon travail au cours de cette thèse.



Vue globale du manuscrit

Ce manuscrit a été construit comme ceci :

L'introduction présente la problématique de ce manuscrit, comme ce qui a été présenté précédemment.

Part I explique le contexte du manuscrit plus en détail.

Chapter 1 décrit l'informatique en nuage de manière générale pour donner un contexte au lecteur. Il donne un aperçu du fonctionnement des applications fonctionnant dans le nuage et de la gestion des infrastructures en nuage afin de donner une base pour comprendre l'approche présentée dans cette thèse.

Chapter 2 présente l'informatique en périphérie et ses défis pour expliquer sur quelles attentes nous avons construire notre approche. Nous avons ensuite défini les principes que nous pensons être nécessaires pour une application placée en périphérie.

Chapter 3 décrit plus en détail comment il est possible d'adapter une application Cloud pour l'Edge et pourquoi les collaborations existantes, requises pour cela impliquent des changements très intrusifs et/ou non génériques.

Part II dépeint l'état de l'art concernant la gestion des applications au niveau de la périphérie, qu'elles soient natives ou qu'elles proviennent du nuage.

Chapter 4 et **Chapter 7** servent d'introduction et de conclusion de l'état de l'art. Le premier présente la manière dont nous avons évalué la littérature et le second présente la comparaison globale.

Chapter 5 présente six approches pour gérer l'infrastructure et, plus important encore, les applications sur celle-ci. Il présente en particulier dans la section 5.4 deux des service meshes les plus connus afin de mieux comprendre leur fonctionnement et leur comportement par rapport à nos exigences.

Chapter 6 montre deux approches différentes pour développer une application Edge-native ou adapter une application existante avec des changements intrusifs.

Part III présente ma propre approche pour amener les applications de au niveau de la périphérie en utilisant une solution de type service mesh.

Chapter 8 explique l'approche théorique pour répondre aux attentes présentées dans la section 3.1.



Chapter 9 présente plus en détail le PoC que nous développons pour correspondre à l’approche sus-mentionnée, en particulier la manière dont la réplication est implantée et comment elle a été testée.

Les chapitres de conclusion présentent des discussions sur notre approche, ses limites, ainsi que différentes perspectives pour l’améliorer.

Résumé de l’approche

Pour faire face à la latence et aux déconnexions, il faut qu’une application utilisée soit déployée entièrement sur chaque site en périphérie, pour permettre l’approche “local en priorité”, qui demande que l’application soit capable de fonctionner de manière autonome sur chaque site. Ensuite, pour permettre un système cohérent, cher aux systèmes distribués pour la mobilité et l’utilisation de l’ensemble de l’infrastructure, il convient de donner à l’application la capacité d’être “collaborative au besoin”. Pour permettre la collaboration entre sites, nous devons être en mesure de manipuler le lieu d’exécution des requêtes.

Comme les applications en nuage peuvent être énormes en terme de code, la solution doit être générique et non intrusive afin d’éviter de traiter les informations de localisation dans le code métier, en dehors de l’application. Enfin, étant donné que l’infrastructure en périphérie est très dynamique (avec des déconnexions et des pannes), et pour permettre aux utilisateurs de décider de l’emplacement de l’exécution de la demande (pour des raisons de confidentialité par exemple), il est important de leur donner la possibilité de le choisir à la demande, de manière dynamique.

L’étude de l’état de l’art a donné des indications sur la manière de satisfaire à toutes mes exigences dans un système P2P entièrement décentralisé, bien qu’elle n’ait pas fourni une solution complète pour chacune d’entre elles.

Un service mesh dédié à la gestion de la géo-distribution et des collaborations entre applications était la solution qui répondait à toutes les exigences pour placer les applications du nuage existantes à la périphérie. Pour permettre cette approche, scope-lang est le DSL qui prend en charge la description de requêtes définies par l’utilisateur, à la demande et à granularité fine.

La modularité des applications du nuage basées sur les services et la façon dont leurs services communiquent entre eux par le biais d’API REST sont les principaux éléments qui ont favorisé notre approche, en permettant la transmission de requêtes grâce auxquelles différentes collaborations sont possibles.



La version actuelle permet trois types de collaborations : *sharing* (le partage), qui utilise une ressource d'un autre site, *replication* (la réplication), qui permet aux utilisateurs de placer des ressources identiques sur différents sites en garantissant qu'elles resteront identiques du point de vue de l'API. Enfin, *cross* (le chevauchement) permet aux ressources de s'étendre sur différents sites. Ces trois collaborations permettent d'utiliser les ressources localement en priorité, et entre les sites si nécessaire. Elles contribuent à réduire la latence, permettent la redondance, la tolérance aux pannes et avec les requêtes définies par les utilisateurs, elles leur donnent la possibilité de choisir finement où leurs requêtes seront exécutées et les ressources à utiliser, ce qui garantit également le respect de leur vie privée, puisqu'ils peuvent sélectionner les sites auxquels ils font confiance. En particulier, la réplication utilise des algorithmes et une logique bien connus pour assurer la cohérence et permet aux utilisateurs de manipuler la réplique la plus proche disponible.

Comme perspectives d'amélioration de ce travail, je donne dans ce manuscrit des indications sur une éventuelle extension possible des collaborations grâce à la généralité de l'approche concernant les ressources. Je présente également la classification des dépendances pour assurer la manipulation correcte des ressources dépendantes. Enfin, pour le long terme, j'ai expliqué comment éviter les partages non valides par l'utilisation d'ownership types (types indiquant la propriété d'une ressource).

Titre : Cheops, une approche externe pour géo-distribuer en périphérie les applications à base de micro-services.

Mot clés : Informatique nuagique, Informatique périphérique, modularité, maillage de services

Résumé : Le passage de l'informatique en nuage à l'informatique en périphérie a modifié les exigences relatives aux applications qui y sont exécutées. Si les applications actuelles de l'informatique en nuage sont extrêmement robustes dans ce contexte, elles n'ont pas été conçues pour faire face aux défis inhérents à l'informatique en périphérie, en particulier les déconnexions et les latences élevées que l'on peut observer entre des sites éloignés. Puisque nous disposons déjà d'applications pour le nuage robustes et au code volumineux, la question qui se pose est la suivante : serait-il possible de les utiliser en périphérie en gérant l'échelle et la distribution géographique ? Pour répondre à cette question, je présente d'abord différentes approches exis-

tantes pour faire des applications fonctionnant en périphérie et les lacunes de ces solutions, tout en conservant les réponses intéressantes à des problèmes spécifiques. A partir de cette étude, je présente la solution construite pour amener les applications du nuage à la périphérie tout en donnant aux utilisatrices le choix du lieu d'exécution de leurs requêtes. Cette solution s'appuie sur la modularité des applications existantes du nuage pour créer une approche ressemblant à un maillage de services qui intercepte les demandes entre les services et les redirige en fonction du langage spécifique à un domaine (DSL) que nous avons créé pour permettre aux utilisatrices de spécifier des collaborations entre plusieurs sites en périphérie.

Title: Cheops, a service mesh to geo-distribute micro-service applications at the Edge.

Keywords: Cloud Computing, Edge Computing, modularity, service mesh

Abstract: The shift from Cloud Computing to Edge computing has changed the requirements for the applications running there. While the current Cloud computing applications are extremely robust in the context of the Cloud, they were not made to face the challenges inherent to the Edge computing, especially disconnections and high latencies that can be observed between far sites. Since we already have robust and huge Cloud applications, the question that remains is: could it be possible to use them at the Edge by managing the scale and the geo-distribution of the infrastructure outside of the business logic? To answer this question, I study different existing approaches

to bring applications to the Edge and identify what is missing in these solutions, as well as keeping track of the interesting answers to specific problems. From this study, I present the solution built to bring Cloud applications at the Edge while giving users the choice of the location for their requests executions. This solution relies on the modularity of these existing Cloud applications to create a service-mesh like approach that intercepts the requests between services and redirects them according to the domain-specific language (DSL) we created that allows users to specify collaborations between different Edge sites.