Khrystyna Fedyuk

Degree in Computer Science and Engineering

Sheik: Dynamic location and binding of microservices for cloud/edge settings

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

Master of Science in
Computer Science and Engineering

Adviser:	João Carlos Antunes Leitão, Assistant Professor, NOVA University of Lisbon
Co-adviser:	Maria Cecília Gomes, Assistant Professor, NOVA University of Lisbon

Examination Committee

Chairperson:	João Ricardo Viegas da Costa Seco
Raporteur:	João Nuno de Oliveira e Silva
Member:	João Carlos Antunes Leitão

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

August, 2020

Sheik: Dynamic location and binding of microservices for cloud/edge settings

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

This work would not have been possible without the help of several people to which I am most grateful. First of all I would like to thank my advisor João Leitão, for his patience and dedication throughout the realization of this work.

My thanks also extends to the Department of Informatics of the NOVA University of Lisbon and the NOVA LINCS research centre, which provided me with the necessary tools and that sparked my interest in the line of work presented here. I would like to thank my colleges and friends, for providing an excellent work environment with laughter that helped me through the toughest times.

Finally I would like to express my deepest thanks to my family for their continuous support and encouragement. In particular to my boyfriend João, who is a constant source of inspiration and motivation.

The work presented here was partially supported by FCT-MCTES, through projects NG-STORAGE (#PTDC/CCI-INF/32038/2017) and the NOVALINCS-laboratory for Computer Science and Informatics research center (UID/CEC/04516/2013).

Experiments presented in this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Abstract

Cloud computing made computing as a utility a reality by providing scalable on-demand resources to service providers. Widely adopted by the industry, cloud computing enabled the success of web and mobile applications. However, the wide use of mobile applications and the explosion of the Internet of Things (IoT) devices are threatening the future success of this business model, as the immense amount of data produced and transmitted by these devices is overloading the cloud servers and stagnating the network paths to cloud infrastructures.

As a possible solution to this phenomenon, edge computing emerged as a distributed computing paradigm that enables applications to leverage on resources outside the boundaries of data centers, bringing computations closer to the end user, and thus provide better quality of service in terms of availability and latency. The resulting hybrid cloud/edge infrastructure however, is highly heterogeneous due to a high number of very diverse edge resources, some of which with limited computational and storage power. As a result, cloud/edge applications must be very dynamic and agile in order to adapt to the constantly evolving execution conditions.

One way to simplify the construction of efficient and robust cloud/edge applications is by leveraging on Microservice Architectures, allowing us to decompose applications into a set of small, independent, and single purpose services that can be developed independently. However, the large number of microservices deployed across an evenly large number of heterogeneous devices becomes too hard to manage, since microservices need to communicate with each other, and in order to be able to interact they need to know each others network locations (IP address and port), a task that is not easily solved when static configuration is not viable, and the management of microservices is performed in an automatic fashion (launching and deactivating instances across the cloud/edge spectrum).

Thus, in this work, we aim at developing a microservice discovery service, called Sheik, that simplifies the interactions between microservices by providing a registration and discovery service that allows to dynamically bind and locate microservices in the cloud/edge settings.

Keywords: Edge Computing, Microservices, Discovery Service, Partial Replication

Resumo

A computação na nuvem tornou a computação como uma utilidade uma realidade, ao fornecer recursos, que podem ser escalados em função da procura, aos fornecedores de serviços. Amplamente adotada pela indústria, a computação na nuvem possibilitou o sucesso das aplicações web e móveis. No entanto, o amplo uso de dispositivos móveis e a explosão dos dispositivos da Internet das Coisas estão a ameaçar o sucesso deste modelo de negócios, já que a imensa quantidade de dados produzidos e transmitidos por esses dispositivos sobrecarregam os servidores da nuvem e estagnam as comunicações da rede.

Como uma possível solução para este fenômeno, a computação na berma emergiu como um paradigma de computação distribuída que permite que as aplicações aproveitem recursos fora dos limites dos centros de dados, aproximando as computações aos consumidores finais, e, como consequência, proporcionar uma melhor qualidade de serviço em termos de disponibilidade e latência. A infraestrutura híbrida de nuvem/berma resultante, no entanto, é altamente heterogênea devido a um grande número de recursos altamente diversos presentes na berma, alguns dos quais têm capacidade computacional e de armazenamento limitada. Como resultado, essas aplicações híbridas devem ser muito dinâmicas para se poderem adaptar às condições de execução em constante evolução.

Um forma de simplificar o desenvolvimento de aplicações híbridas eficientes e robustas, é tirando partido de arquiteturas de microserviços, que permitem decompor as aplicações num conjunto de pequenos e independentes serviços, com finalidade única, que podem ser desenvolvidos de forma independente. No entanto, o grande número de microserviços espalhados por um número igualmente grande de dispositivos heterogêneos torna-se muito difícil de gerir, visto que os microserviços precisam de comunicar uns com os outros, e para poder comunicar precisam de saber a sua localização (endereço IP e porta), uma tarefa que não é facilmente resolvida quando uma configuração estática não é visível, se pretendermos gerir instâncias de microserviços de forma dinâmica e automática.

Assim, neste documento, pretendemos desenvolver um serviço de descoberta de serviços, chamado Sheik, que simplifica a comunicação entre microserviços fornecendo um serviço de registo e descoberta, que permite ligar e localizar dinamicamente microsserviços num ambiente de nuvem/berma.

Palavras-chave: Computação na berma, Microserviços, Serviço de Descoberta, Replicação Parcial

Contents

List of Figures

Chapter 1

Introduction

In today's society, having a smart phone to instantly check the latest news on Facebook or proof check a fact on Google during a discussion with our friends is as natural as breathing and, more and more, user-centric applications are becoming a part of our daily lives. This has become possible, in a significant part, due to the appearance of Cloud Computing [77]. Highly scalable and elastic, cloud computing infrastructures allow dynamic allocation of seemingly unlimited resources, adapting to the (dynamic) needs of applications.

Despite its contributions to distributed systems, cloud computing will not be a viable solution to build reliable, available, and efficient systems soon [40]. The increase in the number of mobile devices and the explosion of the Internet of Things (IoT) applications has led to a significant increase in the amount of data produced and consequent increase in the time required for the cloud data centers to process this data. Another difficulty arises with data transmission itself, as the network capacity is not accompanying the increase in the amount of data transmitted, increasingly becoming a significant bottleneck for timely process of data by cloud infrastructures.

Edge Computing [65] emerged as a strategy to complement cloud architectures and can be broadly defined as performing computations outside of the data centers. It allows to offload some of the computations, such as data analysis and filtering of data sources, to the edge, lowering network traffic and decreasing user-perceived latency. Thus reducing the data deluge over data centers and minimizing the computational load imposed on cloud infrastructures. The edge can be considered as any device with computational power and communication capacity between the source of the data and the cloud data centers. These devices can be ISP servers, (smart) routers & switches, mobile devices, sensors and actuators. It is important to note that, as we progress further from the cloud, the number of devices increases, while the capacity of each individual device decreases

[40].

Due to the large number of very diverse edge resources, the cloud/edge environment is highly heterogeneous and must be extremely adaptive to manage the variable availability and demand of resources, forcing applications to rapidly adapt to diverse execution conditions and sudden changes in the workload [41]. A way to simplify the development and integration of such applications is through Microservice Architecture (MSA) [29], where each microservice is a small, independent, and single purpose service, whose combination allows to define the applications as a whole.

One of the advantages of using MSA is that, as services are independent from each other, they can be developed, deployed, and scaled independently making them very cost effective. Their small size allows them to be effectively replicated/migrated to edge devices where they might have the highest impact on the overall performance of applications, particularly considering user-perceived performance criteria, such as latency.

Problem Definition

When we consider larger scale applications, such as Netflix for example that has hundreds of microservices [56], or many others as represented in Figure 1.1, the large number of microservices deployed across an evenly large number of heterogeneous devices becomes extremely difficult to manage, and impossible to do by human operators in useful time, since microservices need to communicate with each other, and in order to be able to make requests they need to know each others network locations (IP address and port).

Contrary to traditional monolithic applications running on a cloud, whose network locations are relatively static and easily manageable, in MSA applications, particularly those whose components can be dynamically deployed in the cloud and edge, this is a much more difficult challenge to solve, as microservice instances have dynamically assigned network locations. Combined with the variable workload (due to irregular access patterns both in space and time) that requires dynamic reconfiguration of the application, either by moving or partially replicating some of the components, the need of a dynamic discovery service arises, that allows interactions between microservices to be reestricted to closer instances.

To tackle this problem, we need to be able to dynamically register and unregister microservice instances, in order to keep a highly available and up to date registry, and provide information about instance's network locations on demand. Additionally, taking in consideration that microservices are geo-replicated, it is important to provide a load balancing mechanism, integrated in the discovery service, that takes into account microservices locations, so that microservices can communicate preferably with other instances in close vicinity, rather than one that is further away, which could defeat the purpose of having instance in the edge.

Taking this into account, in this thesis, we propose a novel discovery service called Sheik, that is a registration and discovery service that allows to dynamically bind and locate microservices in cloud/edge settings, conseidering the aspects discussed above.

This novel discovery service will be a crucial part into further developing Triforce, a middleware a middleware solution that dynamically and autonomically handles the complexity of monitoring, migrating, and replicating miscroservices in hybrid cloud/edge environments.

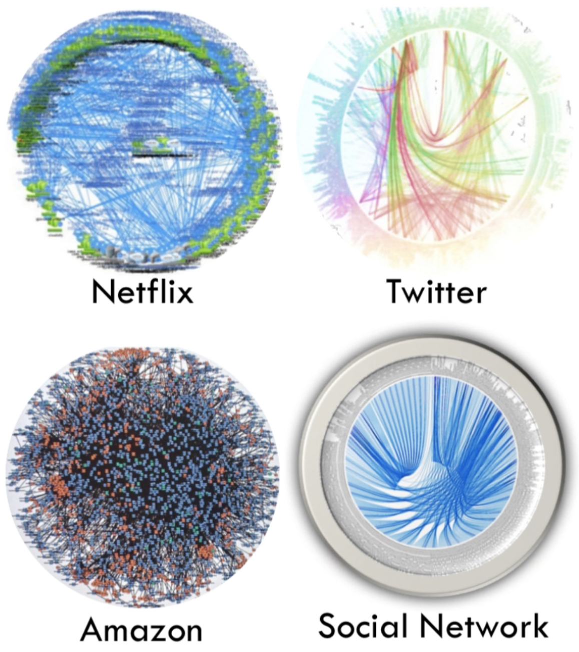

Figure 1.1: "Death star" interaction diagrams of microservice based applications. Adapted from [73].

Contributions

This work provides two main contributions, and can be summarised as follows:

1. A discovery service, named Sheik, that allows the dynamic registration and discovery of microservices in cloud/edge settings.

2. Sheik's comparative experimental evaluation with Eureka, a relevant state-of-the-art solution that serves as a performance baseline to our own solution, in several deployment scenarios (discussed in the nest chapter).

Structure of the Document

The rest of the document is organized as follows:

- In Chapter 2 we present cloud and edge computing in more detail, discuss the properties of Microservice Architectures in comparison with Monolithic Architectures, analyse forms of microservice communication and relevant state-of-the-art solutions, and finally, an overview of replication and information dissemination.
- Chapter 3 details Sheik's design and implementation. We start by presenting the rationale for its design guided by concrete properties that we wanted to achieve, following by its architecture and main components, as well as the details of its implementation.
- In Chapter 4 we present a comparative experimental evaluation of our solution in various deployment scenarios, and a scalabily study.
- Chapter 5 concludes the thesis and provides directions for future work.

CHAPTER 2

Related Work

As our work is going to address hybrid infrastructures that reside both in the cloud and the edge, in this document we will discuss the technologies that exist in the cloud (Section 2.1) and the technologies that exist in the edge (Section 2.2) to be able to make the parallel between the two. Additionally, we will discuss the architecture patterns (Section 2.3) that best fit our solution, followed by a discussion about forms of microservices communication (Section 2.4) and discovery services (Section 2.5) that facilitate this communication. Finally, replication (Section 2.6) and information dissemination (Section 2.7) strategies will be discussed in order to able to extend the applications from the cloud to the edge environment.

2.1 Cloud Computing

Cloud computing appeared as a solution for the long lasting dream of computing as a utility [10]. Providing access to software and data anywhere, anytime, and on any device, cloud computing transformed the Information Technology (IT) industry, making software more appealing as a service and ever impacting how hardware is designed and purchased. Cloud computing offers ubiquitous on-demand access, pay-as-you-go utility-based pricing and virtually infinite resources making it very attractive for business owners that can avoid significant costs with infrastructure and management of these infrastructures. Additionally, cloud computing provides data centers located around the world, enabling a better overall Quality of Service (QoS) by leveraging on Geo distribution and replication.

From a business stand point, cloud computing allows to save a lot of costs as it eliminates the need for service providers to plan ahead for provisioning, dynamically increasing resource pooling when there is a rise on service demand [77].

Nevertheless, cloud computing is a very broad term, as it refers both to the applications delivered as services over the Internet such as Facebook, Twitter, Reddit, or Youtube, and the hardware and systems software, in the data centers, that support these services (Google App Engine, Amazon EC2, GoGrid, etc) [10, 68].

Using an external definition, and in summary, according to The National Institute of Standards and Technology (NIST) [53], cloud computing can be seen as: *a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

2.1.1 Cloud Computing Models

Layered Model

Cloud computing has a modular architecture that can be split into four layers [53, 77]:

Hardware layer: it is typically implemented in data centers and it's responsible for managing the physical resources of the cloud, such as servers, routers, switches, and power and cooling systems.

Infrastructure layer: partitions the physical resources using virtualization technologies, such as Xen or VMware, to create a pool of storage and computing resources. It is a key component in cloud computing, since in a virtualized environment computing resources can be dynamically created, expanded, shrunk or moved on demand. Virtualization provides important advantages in sharing, manageability and isolation.

Platform layer: consists of operating systems and application frameworks, and its built on top of the infrastructure layer to simplify the deployment of applications into the VM machines. For example, Google App Engine, that operates at the platform layer, provides API support to users for simpler implementation of storage, database, and business logic of typical web applications.

Application layer: consists of the actual applications deployed to the cloud, such as GMail, that by leveraging the automatic-scaling feature achieve better performance, availability, and lower operating cost.

As the reader can note from the presentation above, each layer is loosely coupled with the layers above and below it, which allows them to be developed independently. This architectural modularity allows cloud computing to reduce management and maintenance overhead while still supporting a wide range of applications.

Service Model

Cloud computing operates as a service-driven business model where hardware and platform-level resources are provided as services on-demand. Thus, the layers described above can be mapped into three categories, where each layer of the architecture can be implemented as a service (i.e. provider) to the layer above and as a customer (i.e. client) of the layer below [53, 77].

Infrastructure as a Service (IaaS): provides processing, storage, network, and other fundamental computing resources, allowing the consumer to control the operating system, storage and deployed applications. Amazon EC2 and GoGrid are some of the examples of IaaS.

Platform as a Service (PaaS): provides platform level resources, including operating system support and software development frameworks. Examples include Google App Engine, Microsoft Azure, and the Oracle Cloud Platform.

Software as a Service (SaaS): provides on-demand applications, such as Google App Store, Facebook, Amazon Aurora, Amazon DynamoDB, that run on cloud environments over the Internet (typically with users scattered throughout the world).

Types of Cloud Environments

The cloud environment provides different types of clouds, each one with its own benefits and disadvantages, that need to be considered when moving an application to the cloud [18, 77]:

Private Cloud: provisioned for exclusive use by a single organization, typically owned, managed, and operated by the organization itself or by external providers. Often criticized for being similar to a regular proprietary server, private clouds offer users benefits such as high control over performance, reliability, and security.

Community Cloud: provisioned for exclusive use by a specific community of consumers that have shared concerns. Community clouds allows the involved organizations to easily share and collaborate at a lower cost. However, this approach may not be the right choice for every organization.

Public Cloud: provisioned for open use by the general public. Public clouds offers users benefits such as no up-front capital costs and the ability to shift risks to the infrastructure providers. However, public clouds lack to provide fine-grained control over data, network, and security settings.

Hybrid Cloud: composition of two or more distinct cloud infrastructures (private, community, or public), that tries to address the limitations of each approach.

Virtual Private Cloud: alternative solution aimed at addressing issues related to public and private clouds, taking advantage of virtual private network (VPN) technologies allowing business owners to setup their own topology and security settings.

2.1.2 Cloud Computing Characteristics

General Characteristics

Cloud computing provides several important features that allows it to meet the technological and economic demand for Information Technology (IT) [53, 77]:

- Multi-tenancy and resource pooling: the infrastructure provider offers a pool of computing resources that are not allocated exclusively to one client, but rather shared among various clients (one server provides resources to multiple virtual machines of potentially multiple clients). These resources are dynamically assigned to each client according to consumer demand, providing a lot of flexibility to clients for managing their own resource usage and operating costs.
- Isolation: different user applications don't interfere with each other despite running on the same infrastructure (network, and potentially physical machines).
- Geo-distribution and ubiquitous network access: cloud services are available over the Internet and can be accessed through standard mechanisms that support heterogeneity of devices. Which means, any device with Internet connection (such as a smart phone, a laptop, or a workstation) can access cloud services. Additionally, many data centers are located around the globe in order to provide maximum service utility, leveraging on a higher network performance due to proximity to users or client applications (since round trip times are shorter).
- Service oriented: as stated above, cloud computing follows a service-driven operating model. Each layer in the service model offers its service according to the Service Level Agreement (SLA) contractualized with the customers, that it is mandatory to fulfill (with monetary penalties in case of breaches).
- Elasticity: one of the most important features of cloud computing, as it allows resources to be obtained and released, sometimes automatically, according to the current demand. This elasticity is a key aspect that to provide the appearance of infinite resources.
- Self-organizing: as resources can be allocated dynamically, service providers support automatic resource management in order to be able to respond effectively to rapid changes in service demand.
- Utility-based pricing: cloud computing employs a pay-per-use pricing model where customers are charged for what they actually use.

Economical Characteristics

Since its appearance, cloud computing has made a enormous impact on the IT industry and provides several compelling features that make it attractive to business owners [10, 77]:

- No up-front investment: cloud computing uses a pay-as-you-go pricing model, meaning, the business owner doesn't need to invest in an infrastructure. It can simply rent resources according to its own needs and only pay for the (effective) usage.

- Lowering operation cost: since the cloud environment is highly elastic, resources can be rapidly allocated and de-allocated on demand. This allows the business owner to save on operating costs since resources can be released when service demand is low and there is no need to (over) provision capacities according to peak loads.
- Highly scalable: a business owner can easily expand its service to large scales in case of a rapid increase in service demands.
- Easy access: services hosted in the cloud are easily accessed through a variety of devices with Internet connection, since they are usually web-based.
- Reducing business risks and maintenance expenses: the outsourcing of the service infrastructure to the cloud shifts the business risk towards the infrastructure provider, that generally is better equipped and more knowledgeable to manage it. It also allows to reduce maintenance and staff training costs.

2.1.3 Cloud Computing Operators and Services

Although there is a vast amount of services providers for cloud solutions, there are three that effectively dominate the industry, namely Google Cloud Platform (in particular Google App Engine), Microsoft Azure Platform, and Amazon Elastic Compute Cloud (EC2).

All three services offer computation, storage, and auto scaling solutions, but in slightly different ways:

Google App Engine [38] is a Platform as a Service (PaaS) whose target applications are traditional web applications, managed in Google's data centers. One advantage of this platform is that Google deals with all the management overhead and scaling of resources letting the user focus only on building and deploying the applications, without the worry of managing the underlying infrastructure. Google also offers monitoring, logging, and diagnostics tools allowing the user to easily debug and monitor the health and performance of her applications.

Microsoft Azure Platform is also a PaaS and has available a large set of products. One of them, Azure Cloud Services [54], similarly to Google App Engine, handles the provisioning, load balancing, and health monitoring of the applications, as it keeps the applications continuously available during crashes and failures, redirecting traffic from troubled instances to ones that are running smoothly. It also makes available Azure AutoScale, that automatically scales resources up or down to meet the demand, and the user just needs to set the scaling limits.

Amazon EC2 [4] is an Infrastructure as a Service (IaaS) that enables cloud users to launch and manage server instances using APIs. The main advantage of this platform is that it provides bare metal instances in the sense that applications have direct access to the processor and memory of the underlying server, allowing users to have nearly full control of the entire software stack. Another advantage are the flexible storage options available, as beyond the built-in instance storage Amazon also provides Amazon Elastic Block Store (Amazon EBS), which is a persistent, highly available, consistent, and low-latency block storage, and Amazon Elastic File System (Amazon EFS) a simple, scalable, persistent, and fully managed cloud file storage for shared access.

2.1.4 Discussion

Although cloud computing has been widely adopted by the industry, bringing a lot of technical and economical advantages, it is not the perfect solution for building available and efficient applications as new challenges emerge with the evolution of technology, particularly the fast growing Internet of Thing (IoT) and Internet of Everything (IoE) applications.

The increasing amount of data produced by the equally increasing amount of devices with computational and communication capabilities leads to more computation being performed by the data centers to process this data and, consequentially, more communication is performed to transport the data between devices that produce and consume it and cloud infrastructures. As scalable and elastic as cloud computing can be, dealing with such large amounts of data will lead to an increase in the time required to process this data, which in today's society is unacceptable.

According to a study made by Cisco [22], the data globally created by IoE devices will reach 507.4 ZB in 2019, while the annual global IP traffic will only reach 10.4 ZB. This leads us to another problem, the network bandwidth is not increasing at a fast enough rate, becoming a significant bottleneck for the timely processing of data for cloud-based applications.

The result of these problems combined is an increase in user perceived-latency, rise of the load imposed on the cloud services, and an overall degradation of Quality of Service (QoS).

Thus, next we discuss Edge Computing, which is a paradigm that has appeared recently and that attempts, in a way, to mitigate these problems.

2.2 Edge Computing

The success of IoT and mobile applications, which generates large volumes of data at terminals in the outskirts of systems, led to the emergence of edge computing, a new paradigm where data processing occurs, in part, at the edge of the network instead of exclusively in the cloud (i.e, the center, or core of the network). Edge computing complements cloud architectures with computational nodes physically closer to the data sources, thus allowing to move some of the computation closer to the locations where data is both produced and consumed.

Edge computing provides four key advantages:

- minimizes the user-perceived latency by processing the user's request at the edge instead of processing it at a far located cloud data center.
- minimizes the computational load set upon cloud infrastructures by leveraging on edge nodes, and consequentially reduce energy usage by the cloud data centers.
- minimize network traffic by filtering and processing the data at the edge nodes and only send to the cloud the filtered (i.e, processed and essential) information.
- minimize the load imposed on end user devices by offloading some high processing task to nodes in close vicinity at the edge, thus saving limited resources such as battery life.

As a new emerging technology paradigm, edge computing is far from having a standardized definition, architectures and protocols, and various researchers define edge computing in their own way. In this thesis we adopt a generic definition (see Chapter 1) and next will discuss some materializations of this definition.

2.2.1 Edge Computing Variants

In this chapter we will present several approaches that effectively constitute variants of edge computing that are present in the literature and being used in practice now-a-days. We present this variants in a way that is systematically moving away from the cloud.

Lambda Functions

A lambda function is a small piece of code that carries out a specific task. Typically, lambda functions are used as small on-demand applications that are responsive to events and new information.

A relevant example of this technology is AWS Lambda from Amazon [12]. AWS Lambda is an event-driven, serverless computing service that runs code in response to events and automatically manages the computing resources required by that code. Each Lambda function runs in an isolated computing environment with its own resources and view of the file system. Developers can use AWS Lambda to extend other AWS services with custom logic (as Lambda functions can be associated to specific AWS resources, such as a particular Amazon S3 bucket or Amazon DynamoDB table) or to create a new back-end service that operates at AWS scale.

Amazon also enables developers to run Lambda functions at edge locations around the world (that are geographically closer to end users) with Lambda@Edge [13], and thus reduce latency and improve performance.

Cloudlet

The cloudlet concept was introduced in 2009 by Satyanarayanan et al. [66] as a solution to address mobile device's resource limitations, namely their lower amounts of memory and limited battery life when compared to other stationary devices. Cloudlets can be seen as a "data center in a box" whose purpose is to bring cloud services closer to mobile devices. Internally, a cloudlet features a cluster of resource-rich multicore computers with high-speed internet connectivity and a high bandwidth wireless LAN. For safety purposes, the cloudlets contain a tamper-resistant enclosure in order to ensure security in unmonitored areas. The mobile device, that functions as a thin client, offloads computational tasks to the cloudlet through a wireless network, that is deployed close to the locations where devices are used (single hop to access path). The cloudlets physical proximity is very important as end-to-end response time must be very small and predictable. If no cloudlets are in range of the mobile device, it should switch gracefully to the distant cloud, or, in the worst case, rely solely on its own resources (e.g., when no connectivity exists). It is important to note that a cloudlet only contains soft state, such as cache copies of data, thus making it simple in management.

Akamai is the leading provider of cloudlet solutions [60], where each solution extends the application logic to the edge of the Akamai platform. Currently, Akamai provides nine cloudlet services [2]: application load balance, visitor prioritization, edge redirector, phased release, API prioritization, forward rewrite, audience segmentation, request control, and input validation, where each cloudlet has been designed to solve a specific business or operational problem.

Fog Computing

Similarly to how fog, in real life, is a thick condensed water located below the clouds, fog computing takes place beneath the cloud in a layer whose infrastructure connects end devices with the cloud data centers. First introduced by CISCO [35], fog computing tries to bring computation closer to the things (of IOT) that produce and act on data by placing computing capabilities in the network path between the devices and the cloud. This capability is usually put into a device, called fog node, that acts as a gateway that directs different types of data to the optimal place for analysis: *i)* the fog node closest to the device generating the data, if the data is time-sensitive (milliseconds to subsecond response times); *ii)* the aggregation node if the data can wait seconds or minutes for action; or *iii)* the cloud for historical analysis, big data analytics, and long-term storage if the data is less time sensitive (minutes, days, or weeks for the response).

A fog node can be any device with computing, storage, and network connectivity and can be deployed anywhere with a network connection such as a restaurant, on top of a power pole, or even in a vehicle.

Fog computing, due its configuration and infrastructure, brings benefits such as real-time monitoring, actuation, data analysis with reduced latency, improved QoS, and saving of bandwidth as data is processed at the edge of the network [16].

Fog computing has a broad range of applications such as in: *i)* wireless sensor and actuator networks [17], where fog nodes proximity and location awareness can be leveraged to support energy-constrained devices used in these types of applications; *ii)* smart grid systems [76], where the centralized server called SCADA system (that is responsible to stabilize the grid based on status information) can be complemented by a decentralized model with micro-grids that are deployed in the fog infrastructure; and *iii)* smart vehicles, as fog nodes can be integrated into vehicular networks providing either infrastructure-based support such as in VTube [52] where fog nodes are deployed alongside the road, or autonomous support that makes use of vehicles on-the-fly to form a fog-cloud to support ad-hoc events [31].

Mist Computing

The same way fog computing is comparable to its meteorological counterpart, so is mist computing. Defined as the thin layer of floating water droplets located on the ground, mist computing, an evolution of fog computing, takes place "on the ground" at the very edge of the network, where the light computing power is located in devices that can be sensor and actuator nodes. Mist computing pushes the computation towards the sensors in IoT applications, which enables the sensors themselves to perform computations, in particular, data filtering, avoiding to disseminate non-relevant data and thus, alleviating some of the load imposed on the fog and cloud servers.

However, the computing power of mist computing comes from the microchips or the micro-controllers embedded on these devices, which makes their processing capabilities very limited.

Similarly to fog computing, wireless sensor and actuator networks can leverage on mist computing by offloading some of the computations, such as data filtering, to the sensors. Another application of mist computing is in smart houses, as with the rapid development of IoT devices more and more smart devices and sensors are connected at homes and can be used to pre-process some of the data before sending them to fog nodes or cloud infrastructures.

Mobile Edge Computing

Mobile Edge Computing (MEC), in its standardized version, is a paradigm that aims to bring cloud computing functionality to the mobile edge. MEC offers lower latency, proximity, context and location awareness, and higher bandwidth by deploying MEC servers at cellular base stations [16]. These MEC servers act as cloud servers running at the mobile edge where the time sensitive part of applications can be executed, whereas delay tolerant compute intensive part of application is forwarded to cloud servers. Thus, instead of forwarding all the traffic to the cloud servers, MEC shifts some of the traffic to the MEC servers allowing them to run applications and perform computations closer to the mobile devices, what results in reduced network saturation and faster response times.

One of the main problems with the standardized MEC architecture however, is that in a very crowded environment, such as a stadium or a concert, the infrastructure behind the MEC servers becomes overloaded, acting as a bottleneck and significantly degrading the QoS. Due to this, some researchers, as in case of Drolia et al. [30], see MEC as an "edge-cloud" consisting of a cluster of mobile devices that collectively pool their storage and computation resources, over a local network, towards a common goal. This version of MEC takes advantage of the high density of mobile devices and the increasing amount of resources each new version of these devices contains. In this approach, a collection of mobile devices collaboratively acts as a (shared) computational resource, thus being able to perform computationally heavier tasks without resorting to cloud data centers.

Serendepity [67] is a framework that tries to materialize this concept by distributing computation tasks among other nearby mobile devices in order to speed up computation or to save energy, where a mobile device may execute tasks locally or remotely on other available devices based on an optimization criterion. In [59] the authors try to solve both computation offloading and context adaptation challenges by developing CloudAware, a framework whose goal is to support ad-hoc and short-time interaction with not only centralized resources but also nearby devices, and to provide an uninterrupted availability of the application even if no other devices are available or the connection gets interrupted by using the mobile device as a fallback. On the other hand, Thyme [69] is a time-aware reactive data storage system that addresses the challenge of supporting reliable and efficient data storage and dissemination among co-located wireless mobile devices by exploiting the synergies between the storage substrate and the publish/subscribe paradigm without resorting to centralized services or network infrastructures.

2.2.2 Discussion

As it is common with all new technologies, the presented architectures all have its flaws. Fog computing and MEC, in particularly, have a serious bottleneck limitation because they rely on the infrastructure (the gateway device in the case of Fog computing and cellular base stations in case of MEC) for the whole system to operate, thus presenting a single point of failure. Although, recent proposals such as [69] are exploring how to offer distributed abstractions on MEC contexts while lowering the dependence of cloud and network infrastructures. Mist computing on the other hand, faces challenges related to the heterogeneity of the sensor devices and their limited processing power, which makes implementing solutions significantly more complex. Cloudlets have limitations related to dimensioning, in the sense that how much processing, storage, and networking capacity should they have in order to be able to provide satisfactory service while still being cost effective.

In addition, while the architectures described above all explore the potential of edge computing in some way, they do so in a limited way, as they require specialized hardware and do not take advantage of the devices that already exist in the edge. Most importantly, all these architectures are optimized for IoT and mobile devices and do not present a generic solution. In this thesis, we argue that edge computing, when considered in a more holistic way, offers the potential to build new edge-enabled applications, whose usage of edge resources goes beyond what has been done in the variants discussed here. These edge-enabled applications must be highly interactive and have very short response times, as they will have a human end user interacting with the application, contrary to the typical IoT applications where the end user is usually out of the main computation loop and only receives information in a feed through cloud infrastructures. The setting of these applications is also distinct, in the sense that it is intended to take advantage of the resources that are already available in the edge, thus, in a way, unify the variants presented above.

Another more conceptual flaw of edge computing is that it has to be highly flexible to cope with the variable demand and availability of resources, as it heavily depends on how many clients there are in each location and what kind of tasks they are performing. This implies that the applications that run on the edge must be able to transition naturally from the cloud architecture to the edge as needed, and when it is no longer necessary to be deactivated without losing persistent state, in order to avoid consuming resources at the edge. This of course implies that, one must study the current architectural patterns for developing user-centric applications that enable the migration of (generic) computational tasks among different (compatible) edge resources, and that also allow computations to be decomposed into simpler computational tasks dynamically.

2.3 Architecture Patterns

Architecture patterns are informal and idiosyncratic descriptions of a system [55] and, when implemented by a system, provide insight of key aspects of a system such as scaling and portability, the assignment of functionality to design elements, interaction protocols between elements, and global system properties such as processing rates, end-to-end capacities, and overall performance. Architecture patterns provide a level of abstraction that allows developers to reason about a system behaviour (function, reliability, and performance) and, by characterizing the crucial system design assumptions, facilitate to reason about what aspects of the system can be easily changed without compromising system integrity.

Although there are several architecture patterns studied and discussed in the literature, in the scope of this thesis, we will focus mainly on two: monolithic architecture and microservices architecture, that we now discuss in more detail.

2.3.1 Monolithic Architecture

Monolithic Architecture is a traditional approach to build applications in which the application is built, packaged and deployed as a monolith, i.e., a single executable artefact. Although monolithic applications have a logically modular architecture, these modularisation abstractions rely on the sharing of resources of the same machine, and thus are not independently executable [29]. An application developed using a typical monolithic approach, has a single codebase and would be developed using an MVC framework such as .NET, Rails, Spring, among many others, for example [74], as these frameworks allow to develop three tiers applications providing different tools and libraries to develop the presentation, business and persistence layers.

Applications written following this architecture are very common, as it provides several advantages: *i)* monolithic applications are simple to develop since our IDEs and other tools are focused on building single applications; *ii)* monolithic applications are simpler to test, as one can implement end-to-end testing by simply launching the application and testing the UI with tools such as Selenium or JMeter; *iii)* monolithic applications are also simple to deploy, as it is simply necessary to upload the packaged application to a server; and *iv)* monolithic applications are easily scalable horizontally by running multiple copies behind a load balancer. Overall, monolithic applications make it simple to develop, deploy, test and scale applications when the size of the codebase is relatively small, which leads to monolithic applications being easier to develop at the beginning of a project.

Nevertheless, monolithic applications have their downsides when the application codebase grows big and the changes have to be made rapidly [39], as as applications grow in size they become harder to maintain and make changes to them. Another downside is that monolithic applications suffer from "dependency hell", where adding or updating libraries results in inconsistent systems that do not compile/run or misbehave. Additionally, fine-grained scaling is impossible with monolithic applications, because the whole application needs to be deployed every time.

2.3.2 Microservices Architecture

One of the more recent venues being explored towards achieving flexible installations and execution is the transition from monolithic architecture to microservices architecture. Microservice Architecture (MSA) [29, 47] is an emerging paradigm to develop, deploy, execute, and manage applications that can be seen as an evolution of the Service Oriented Arquitecture (SOA) [61]. Already adopted by the streaming giant Netflix [75] and LinkedIn [36], MSA defines applications as a set of small, independent, single purpose services, each running its own processes and communicating via light-weight mechanisms. This enables an easier development and integration of components with emphasis on the design and development of highly maintainable and scalable software, since each microservice can be developed independently. Additionally, each individual microservice has an independent life-cycle and relies on its own independent data storage solution to manage its state [37].

All these qualities make microservices a more desirable option than the traditional monolithic applications as it offers key advantages over them [39]: *i)* scaling microservices is easier than scaling a monolithic application, since one can deploy just the number of instances of each service that satisfy its capacity and availability constraints instead of scaling whole application and deploy the whole codebase; *ii)* microservices tackle the problem of complexity as they decompose what would otherwise be a large monolithic application into manageable services, that are much faster to develop, and easier to maintain and evolve.

2.3.3 Discussion

Considering the properties of each architectural style, it is clear that MSA should be leveraged in the development of cloud/edge applications as it would greatly simplify the design, development, and management of these applications mainly because: *i)* as we want to replicate the application logic across the edge nodes with fine-grain granularity, if the application is already decomposed into microservices that are self-contained, this greatly facilitates replication; and *ii)* microservices can be efficiently moved/replicated to where they might have the most impact on performance, since usually they are relatively small software components.

In a setting where the idea is that microservices are constantly being deployed and migrated, communication between microservices is key to ensure that the overall performance of the application can benefit from the continuous evolving deployment, as otherwise it can lead to a situation where microservices lose their autonomy (being dependent of instances created in cloud for instance) and thus the main benefits of the whole approach can diminish. Thus, in the next section we will discuss the main forms of communication between microservices.

2.4 Microservices Communication

In contrast with a monolithic application, a microservices-based application is a distributed system running on multiple machines, where each service instance is typically an independent process. Consequently, microservices must interact using an inter-process communication (IPC) mechanism. When selecting an IPC mechanism for a microservice application, it is useful to think first about how services interact, as there is a variety of interaction styles. Mainly IPC mechanisms can be categorized into two categories [63]: Publish/Subscribe and Request/Response.

2.4.1 Publish/Subscribe

Publish/Subscribe is a messaging pattern where publishers (senders of messages) categorize messages into events (that can be topic-based, content-based, or hybrid) and publish them to an event manager, without sending them to anyone in particular. Subscribers express their interest in a topic (or type of content, or both) in order to be notified subsequently of any relevant event, generated by a publisher [34]. The standard system model for publish/subscribe interaction relies on an event notification service (typically materialized by one, or a set of, brokers), that stores and manages the subscriptions and delivers the events to the (appropriate) subscribers.

Publish/Subscribe approaches are very useful as they allow to decouple publishers from subscribers, in the sense that publishers don't know how many and who are the subscribes and the subscribers don't know who are the publishers. Additionally, publish/subscribe systems can offer reliability for the communication among publishers and subscribers, and message buffering, where a message broker queues up the published messages until they can be processed by the subscriber, eliminating the need for both instances (the publisher and the subscriber) to be available for the duration of the exchange. There are a few systems providing this abstraction that have become quite popular and used in industry: RabbitMQ [62], Apache Kafka [6], and Azure Service Bus [14].

A particularity of publish/subscribe communication is that it is asynchronous, thus the publisher does not block waiting for a reply. Thus, microservices need be developed assuming that the reply will not be received immediately.

2.4.2 Request/Response

Request/Response is a synchronous communication mode, where a client sends a request to a service, the service processes that request and sends back a response. It is synchronous because the client that makes the request waits for a response, that it expects the request and reply be delivered in a timely fashion, and in a thread-based application, the thread that makes the request usually blocks while waiting. Because the interacting processes assume the roles of client and server, this mode is also referred to as client/server interaction.

Typically, request/response communication is done over HTTP with the help of protocols such as REpresentational State Transfer (REST) [64], an architectural style that defines a set of constraints to be used for creating Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP, and Thrift [7], a framework for writing scalable cross-language remote procedure call (RPC) services.

The advantage of using a communication protocol that is based on HTTP is that HTTP is simple and familiar, as every language offers HTTP support. HTTP is firewall friendly and easily testable from within a browser using an extension such as Postman or from the command line using curl. Additionally, HTTP eliminates the need of an intermediate broker, which simplifies the system's architecture.

2.4.3 Discussion

When comparing both communication strategies described above, a lot of obvious challenges emerge when considering the use of publish/subscribe alternatives [63]: *i)* the messaging system used in publish/subscribe introduces additional operational complexity, as it is yet another system component that must be installed, configured, and operated; *ii)* the messaging system must also be highly available, otherwise system reliability is impacted; *iii)* scalability can also be an issue, as it is not enough to simply replicate services, the messaging system needs be able to deal efficiently with the increasing amount of messages; *iv)* typically web services communicate in a request/response-based interaction, which is very complex to replicate in a publish/subscribe setting, introducing additional workload for developers; and *v)* as data owned by each microservice is private to that microservice, data management is also a more complex task, in the sense that achieving atomicity can be difficult, since communication in publish/subscribe is asynchronous

Due to his, we believe that a classical approach to communicate through HTTP is a better option. Nevertheless, the request/response model is not without flaws, as in order to communicate with each other services need to know the location (the URL) of each other in order to be able to make requests. In a setting where microservices are constantly dynamically replicated and migrated, this is not a task that is simply solvable with a configuration file. Thus clients need to rely on a discovery service, that deals with this operational aspect for them.

2.5 Discovery Services

In order to communicate with each other, microservices need to know the addresses (IP and Port) of other relevant microservices instances. In a simple monolithic architecture, a static configuration is sufficient, as each service is deployed at the same location and this rarely changes. But this is not the case in microservices architectures, as services are deployed independently, and in our case on-demand across various different locations, which leads to the location of different microservices instances to change frequently. Thus a dynamic solution is required.

This dynamic configuration, containing the location of available services, can be maintained by a discovery service, and when a microservice is deployed it will need to register itself in such service. Then, each microservice only needs to know about the location of the discovery service and will use it to get the location of other microservices.

There are many implementations suitable for service discovery (although it has to provide two essential features: fast response times and high availability, the latter through replication typically), that can be divided in two categories, depending on how they perform the load balancing: client-side discovery pattern and server-side discovery pattern [71].

Client-Side Discovery Pattern When using this pattern, the client queries a service registry for available instances and is then responsible to use a load-balancing algorithm to select one of the available service instances and make a request. This pattern is relatively straightforward to use, and since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions. On the other hand, one of the drawbacks of this pattern is the high coupling between the client and the service registry.

Server-Side Discovery Pattern Contrary to the previous pattern, here the client makes a request to a service via a load balancer, that, in its turn, queries the service registry and routes each request to an available service instance. One of the benefits of this pattern is that the details of discovery are abstracted away from the client, since they simply make requests to the load balancer. This in turn leads to an ease of communication and failure safety, as in the case of failure, microservices can be easily restarted, due to their stateless properties. On the other hand, if not designed correctly, the registry could be the main bottleneck of the system, or even a critical point of failure for the system.

2.5.1 Relevant Service Discovery Services

Next, we are going to present some of the current relevant solutions that allow service registry and discovery.

AWS Elastic Load Balancer

AWS Elastic Load Balancer (ELB) [11] is a load balancing solution for edge services exposed to end-user web traffic, that automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, IP addresses, and Lambda functions. Although ELB is commonly used to load balance external traffic from the Internet, it can also be used to load balance traffic that is internal to a virtual private cloud (VPC) making it a suitable solution for microservices applications.

ELB provides high availability, by distributing incoming traffic across a region, in multiple availability zones, ensuring only healthy targets receive traffic. Additionally, ELB allows to use IP addresses to route requests to different targets, this in turn offers clients flexibility in how to virtualize application targets, allowing to host various applications on the same instance.

Apache ZooKeeper

Apache ZooKeeper [9] is an open source project that provides a centralized service for providing configuration information, naming, synchronization and group services over large clusters in distributed systems. The goal is to make these systems easier to manage with improved, more reliable propagation of changes. ZooKeeper aims at distilling the essence of different services in a system into a very simple interface to a centralized coordination service. The service itself is highly reliable and provides implementations of consensus, group management, and presence protocols so that the applications do not need to implement them on their own.

The way Zookeeper allows distributed systems to coordinate with each other is through a shared hierarchical name space of data registers (called znodes), that behave similarly to a file system, but unlike a normal file system, ZooKeeper provides high throughput, low latency, highly available, and strictly ordered access to the znodes. These performance aspects combined with reliability aspects prevent Zookepper from becoming the single point of failure in big systems, allowing it to be used in large distributed systems efficiently.

etcd

etcd [33] is an open source distributed key-value store used to hold and manage the critical information of a distributed system. Most notably, etcd is known as Kubernetes' (the popular container orchestration platform) primary datastore used to store its configuration data, state, and metadata. Because of this, etcd is fully replicated, every node in an etcd cluster has access the full data store, highly available, since it was designed to have no single point of failure and gracefully tolerate hardware failures and network partitions, and reliably consistent, as every read returns the latest write across all clusters.

etcd is written in Go and uses the Raft consensus algorithm [58] to ensure data store consistency across all nodes for a fault-tolerant distributed system. In Raft consensus, leaders handle all client requests which need cluster consensus (typically write operations). A leader is an elected node that manages replication for the other nodes in the cluster, called followers. Once the leader has confirmed that a majority (a quorum of (n/2)+1 nodes,where n is the total number of nodes in the cluster) of follower nodes have stored the new request as a log entry, it applies the entry to its local state machine and returns the result of that execution to the client. If followers crash or network packets are lost, the leader retries until all followers have stored all log entries consistently. However, the client doesn't need to know which node is the leader to be able to send requests. Any request that requires consensus sent to a follower is automatically forwarded to the leader. Requests that do not require consensus (e.g., serialized reads) can be processed by any cluster member. It is important to note that, each cluster can only have one leader at any given time.

Consul

Consul [23] is a distributed, highly available, datacenter-aware, service discovery and configuration system. Like etcd, Consul is written in Go and uses a distributed key-value store based on the Raft algorithm to store its data, but the similarities stop here. Consul is a complex system that has many different moving parts: support for multiple datacenters, two different gossip pools, and a Raft based consensus protocol.

In Consul (Figure 2.1), each datacenter has a mixture of clients and servers, called agents. All the agents that are in a datacenter participate in a gossip protocol, called LAN, that serves three main purposes: *i)* membership information, as it allows clients to automatically discover servers, reducing the amount of configuration needed; *ii)* distributed failure detection, allowing the work of failure detection to be shared by the entire cluster instead of being concentrated on a few servers; and *iii)* reliable and fast event broadcasts, serving as a messaging layer to notify when important events such as leader election take place. On the other hand, the WAN pool, the other available gossip pool, is globally unique as all servers (and only the servers), regardless of the datacenter, participate in the WAN pool. The WAN pool is optimized for the higher latency of the Internet and its main purpose is to allow datacenters to discover each other in a low-touch manner, and because all servers are operating in this pool, it also enables cross-datacenter requests.

In general, data is not replicated between different Consul datacenters, as the servers of each datacenter make part of independent Raft sets. When a request is made for a resource in another datacenter, the local servers forward an RPC request to the remote servers for that resource and return the results.

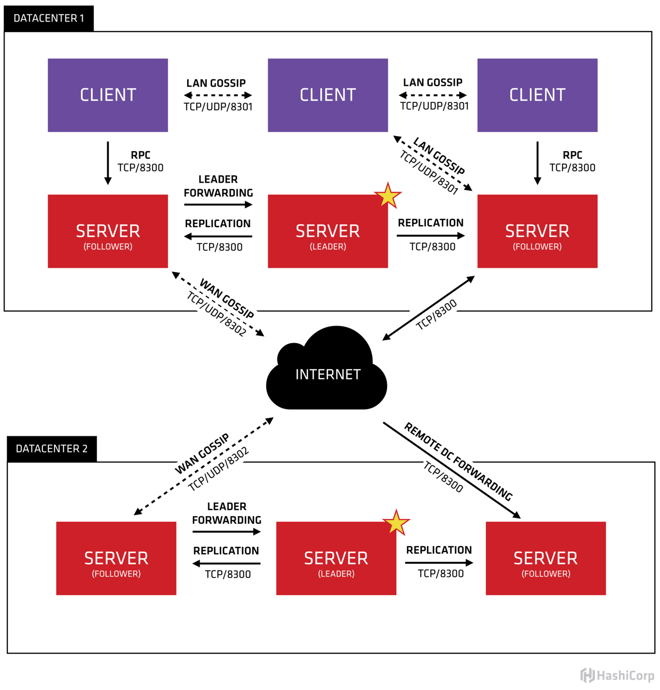

Figure 2.1: Consul's architecture. Adapted from [23].

Eureka

Eureka [57] is the Netflix's Service Discovery Service, composed by two components: Eureka Server and Eureka Client, that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. Basically, Eureka acts as a service registration and discovery service where dedicated servers are responsible to maintain the registry of all the microservices that have been deployed and removed, and whose architecture is depicted in Figure 2.2. When a microservice registers itself with the discovery server, also known as Eureka Server, it needs to provide metadata such as host, port, and health indicator and, in this way, allowing for other microservices to discover it. In order to avoid keeping a registry of dead instances and maintain a stable ecosystem, the discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry. Additionally, Eureka servers will enter self preservation mode if they detect that a larger than expected number of registered clients have terminated their connections in an ungraceful way (failed to send three consecutive heartbeat messages), and are pending eviction at the same time. The self preservation mode triggers when 15% of its registered instances are pending eviction because of unclean terminations, and is done to ensure that temporary catastrophic network events do not wipe out eureka registry data.

To provide better resiliency, Eureka clients are built in a way to handle the failure of one or more Eureka servers. For this, Eureka clients fetch the entire registry information from the server and cache it locally, and then use this information to find other services. The cached information is then periodically updated (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one. After getting the deltas, Eureka client verifies the validity of the information stored by comparing the instance counts returned by the server, and if the information does not match for some reason, the whole registry information is fetched again.

It is important to note, that although Eureka is a distributed service and there is one Eureka cluster per region and at least one Eureka server per zone (to handle zone failures), Eureka replicas in the same region only know about instances in its region, as information is not replicated between regions.

2.5.2 Discussion

Although all of the services described above are interesting solutions for handling service discovery, they are all well though to be deployed in cloud datacenters, having serious limitations when transitioned to an edge setting, since most edge devices present some form of limited computational and storage capacity.

A common problem between these solutions is the total data replication. As stated above, edge devices have limited storage capacity, and a setting with hundreds of microservices spread across various locations would generate a lot of information that would be

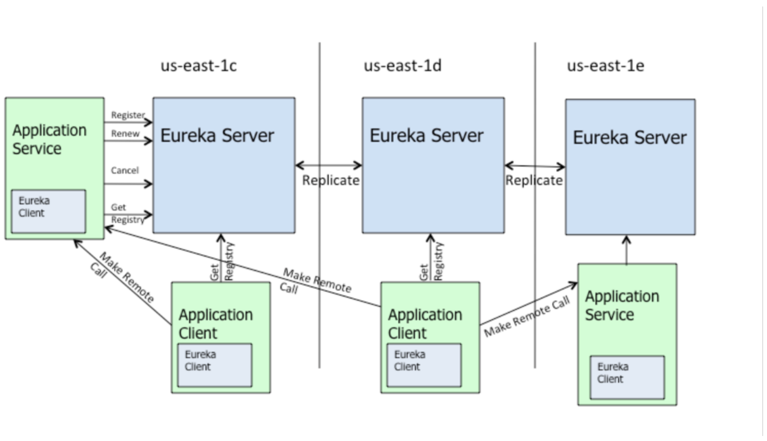

Figure 2.2: Eureka's architecture. Adapted from [57].

unfeasible to store in an edge device, most of which would be unused (if we were to load balance requests based on proximity). An exception to this is Consul, as it doesn't replicate data between clusters but only within a cluster, but this leads to another problem where in order to provide availability and fault tolerance one would need to deploy at least three nodes per every new cluster (in order to be able to achieve majority and not loose data in case of failure). This again, can be difficult to achieve with limited computational power, and as there is total replication between nodes in the cluster the problem with storage capacity arises again.

Additionally, consensus based protocols do not scale well, as although larger clusters provide better availability and fault tolerance, the write performance suffers because data must be replicated across more machines, and its performance suffers with geo-replication, as there is higher consensus request latency from crossing data center boundaries.

etcd performance will particularly suffer in edge settings, as etcd's performance is heavily dependent upon storage disk speed, as etcd persists all stored data to disk. If we were to deploy etcd on an edge device, its performance would be greatly diminished due to edge devices having limited computational capabilities.

Since data replication seems to be the biggest problem among all of the solutions, we will now study data replication as a way to propose a better and viable solution.

2.6 Replication

In its essence replication entails having multiple copies of some resource across multiple processes. That resource can be data, where the same data is stored across multiple devices, or application logic, where the same logic is executed by multiple devices. Replication is fundamental for the effectiveness of distributed systems as it allows applications to provide: *i)* fault-tolerance, as the service always guarantees strictly correct behaviour despite a certain number of system failures, in particularly durability, because if some replicas fail the system does not lose information, and high availability, in case of system failures or network partitions, clients can still interact with the system and modify its state; and *ii)* enhanced performance, as it allows load distribution across multiple machines, which is essential for scalability, and proximity between clients and a resource replica [26].

A replication algorithm is the one responsible for managing the multiple replicas and it's what will restrict what are the effects of operations observed by clients given the story of the system (and potentially their past operations). A valid replication algorithm should provide two key properties: *i)* transparency, meaning the client should not be aware that multiple replicas exist and only be able to observe a single logical state; and *ii)* consistency, in the sense that despite the individual state of each replica, the state that can be observed by a client given its past (operations executed by that client) and the system history (operations executed previously by any client) is restricted.

In the following section we will discuss the various models for data and logic replication.

2.6.1 Replication Models

We consider a total of six models for replication, that can be split into three dimensions, and each one of them has its own properties and performance trade-offs.

First Dimension

The fist dimension deals with which replicas can execute operations (that modify the state of the system):

Active replication: all replicas can execute operations, with the state being continuously updated at every replica (which might lower the impact in case of a replica failure). There is a constraint with this type of replication, as it can only be used when the operations are deterministic, meaning they do not depend on non-deterministic variables such as local time or random values. Additionally, if operations are not commutative then all replicas must agree on the order in which operations should be executed across all replicas.

Multi-Master replication: similar to active replication, except there isn't a constraint on non-deterministic operations. Due to this, additional coordination protocols are required to deal with conflicting operations (which introduces additional overhead, since this protocols tend to be expensive).

Passive replication: operations are executed by a single replica and then the results are sent to other replicas. Although this type of replication deals very well with non-deterministic operations the load across replicas is not balanced, since only one replica effectively deals with computations. Sometimes, this model of replication can be called *Single Master*, when referring to a opposite model of Multi-Master replication.

Second Dimension

The second dimension deals with when the replication process occurs in relation to the reception of the client operations:

Synchronous replication: replicas are updated before the client gets a response. This allows the client to be sure that if he got the answer, the effects of the operation are not going to be lost, but at the cost of higher latency, since this process significantly delays the transmission of the reply for the client.

Asynchronous Replication: replicas are updated sometime after the client obtains a reply or concurrently with the reply being sent to the client. This has the opposite effect of the synchronous replication, as the client will receive replies more quickly, since only one replica needs to update their state to reply, but the effects of a client operation may be lost, even if there was a reply, due to the failure of specific replicas before the synchronization process finishes.

Third Dimension

The third dimension deals with how much/which part of the data/logic to replicate:

Total Replication: the whole application logic or data is replicated across all replicas. Every replica will behave the same way as other replicas, as every update will be propagated to and applied in every single replica, allowing users to access data from any replica.

Partial Replication: only carefully selected portions of the application logic or data is replicated. In some cases, different portions can be replicated in different replicas. Partial replication increases the scalability of replicated systems since updates only need to be applied to a subset of replicas, allowing replicas to handle independent parts of the workload in parallel. Such solutions however, may induce additional overhead to deal with problems such as dependencies among (different) replicated portions.

As stated above, replication can be applied to both the data layer and the application logic layer. Below we will explain how these different concepts map to each of them.

2.6.2 Application Logic Replication

An application can be split into several logical components that perform different tasks for handling different operations, that can be materialized through the use of microservices as we explained in Section 2.3.2, and cooperate with each other in order to provide the application service as a whole. These components can then be individually replicated (partial replication) not only in the cloud but also on edge nodes allowing them to take advantage of the edge infrastructure. Components that are on the edge can filter data, redirect requests, or actually execute requests. For some of them to be able to execute requests they might need access to data, which can lead to the need of replicating the relevant portions of data as well. Since components need to interact with other components, whether it is because of dependencies or simply because one requires access to results computed by another component, it is necessary to keep track of the locations of different instances of the components through a service discovery service. In order to ensure efficient communication, as these components will need to frequently communicate with the discovery service, it may be beneficial to replicate them together to locations in close vicinity, as to promote low latency access.

2.6.2.1 Relevant Application Logic Replication Strategies

Next, we are going to present some solutions that allow to replicate the application logic.

Cloud Elasticity

Cloud computing and its elasticity property is a very important and key element to the success of this architecture. The ability to scale up and down resources in response to the service demand it's what makes cloud computing so appealing.

Although limited to each individual data center, cloud's elasticity is a form of replication, where application logic and sometimes data is replicated according to various metrics, sometimes defined by the user, to maintain the QoS when there is a spike on the load imposed on the service. This granularity control allows service providers to maintain a steady service quality while minimizing costs for clients by decommissioning replicas when the load diminishes.

Geo-Replication

Geo-replication is another form of replication, where the same application logic is scattered around the world in various datacenters. Geo-replication, allows clients to contact the closest datacenter, avoiding the penalty of long round trip times to reach remote datacenters (i.e, in other continents), providing lower-latency. Additionally, Geo-replication provides fault-tolerance and availability, as if a data center fails its requests are redirected to another data center. In Geo-replication the entire application is usually replicated (including logic and data) rather than just the necessary components, and it is usually based on a static deployment, as one does not add data centers on demand or in reaction to some external or unplanned event.

Web Applications

In order to enhance web applications performance, application logic, such as scripts, are often cached and stored in the web browser's memory. This also is a form of application logic replication and its widely used in web development to reduce the loading time of web applications. Nowadays dynamic generation of web pages, where pages are generated on the fly, is broadly used by companies such as Amazon and Facebook to deliver custom ads to users [70]. However, dynamic generation of a web page typically requires issuing multiple queries to a database, so access times to the database can easily become too high when the request load is high. Thus, techniques such as web page caching are used, where fragments of the HTML pages the application generates are cached to serve future requests [21]. These features can also be implemented by leveraging lambda functions, that can inject content to web pages depending on the location of the user.

2.6.3 Data Replication

As application logic components can be migrated and replicated, so does application data might need to be migrated and replicated as to ensure the continuous and efficient operation of components that frequently manipulate the application state. However, hosting a full copy of a potentially large data base for every logic component instance is very inefficient and sometimes even impossible, thus creating partial replicas of the application state (i.e, data) is essential to support partial replication of the application logic on the edge.

However, replicating data can be tricky, as it implies additional challenges related with consistency, namely the data replicated across multiple locations should have the same values and rules should be defined to what data is exposed to clients after replicas are no longer consistent. There are two main families of consistency models, that can be provided: strong and weak.

Strong consistency models enforce the system to provide a strict notion of evolution of the state across all replicas (and thus atomicity) and it's mainly concretized by two models: *i)* linearizability, that provides a guarantee about single operations on single objects. Particularly this consistency model states that once a write completes, all later reads return the value of that write or the value of a later write. Furthermore, once a read returns a concrete value, all later reads should return that value or the value of a later write; and *ii)* serializability that provides a guarantee about groups of one or more operations over one or more objects (usually called transactions). Unlike linearizability, serializability does not impose any real-time constraints on the ordering of operations (it only requires a single order across all replicas). Although strong consistency is preferred over weak consistency, implementing a system that is strongly consistent is both challenging and costly, as the algorithms that implement these consistency models are often complex and nuanced. The most important downfall of strong consistency however, is the high-latency and low availability that comes with this model, that has been captured by the CAP Theorem [19]. According to the CAP Theorem, it is impossible for a distributed system to provide all of the following guarantees simultaneously: consistency, availability, and partition tolerance.

As a way to circumvent these difficulties, systems often abandon strong consistency in favor of weak consistency. Weak consistency models are less strict than strong consistency models as implementations often allow operations to only be executed in a small set of replicas and then further synchronized between them in the background. Being an increasingly popular type of consistency, weak consistency is offered by many concrete models, where the most famous are the eventual consistency model, the weakest form of weak consistency where an eventually consistent system only guarantees that, eventually, when no write operations happen, all the system's replicas will converge to the same state, and the causal consistency model, that enforces that each client always observes a system state that respects the cause/effect relationships between write operations, typically captured by the combination of four fundamental properties: read your writes, monotonic reads, monotonic writes, and writes follows reads. Both these models have their flaws, as in eventual consistency the user may at some point observe inconsistent state until replicas converge, and in causal consistency the replicas may never converge. Due to this, the causal+ consistency model was proposed as a combination of the eventual and causal models guaranteeing both their properties, the client never sees any state that is incompatible to his own causal history and, when no write operations happen, the system will eventually converge to a common state.

There are many solutions that implement the described consistency models. Often the guarantees one wants to provide greatly influence the design of these solutions, leading to different solutions providing different trade-offs between cost, latency, and availability. Thus next, we are going to provide examples from the literature of recognized solutions with different guarantees and will explain both the solutions and the guarantees they provide.

2.6.3.1 Relevant Data Replication Strategies

We are going to present five different data replication systems that provide different levels of consistency, beginning with the weakest and moving up to the strongest.

Dynamo

Dynamo [27] is a highly available distributed key-value storage system built purposely for Amazon's platform that needed to provide reliable, efficient, and highly available service to customers that existing solutions at the time of its creation could not provide. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios, falling under the eventual consistency model. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a simple novel primary-key only interface for developers to use.

Dynamo uses (and introduced) a combination of now-a-days well known techniques to achieve scalability and availability: *i)* data is partitioned and replicated using consistent hashing; *ii)* consistency is facilitated by object versioning with the help of vector clocks; *iii)* consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol; *iv)* membership and failure detection is achieved with a gossip based protocol; and *v)* recovery from permanent failures is solved with anti-entropy using Merkle trees. Additionally, Dynamo is a completely decentralized system with minimal need for manual administration, as storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution of data among different nodes.

COPS/EIGER

Clusters of Order-Preserving Servers (COPS) [50] is a causal+ consistent key-value storage system. COPS was designed with the goal of supporting complex online applications that are hosted on a small number of large scale datacenters, where each datacenter hosts a local COPS cluster with a complete replica of the data. Clients communicate only with their local COPS cluster running in the same datacenter, which allows COPS to execute all read and write operations in a linearizable fashion. Then this data is replicated in the background across the rest of the datacenters according to the causal+ consistency model. These consistency properties come at low cost as the performance and overhead of COPS is similar to prior systems while providing much greater scalability.

EIGER [51] is a scalable geo-replicated data store that provides guaranteed low latency, a rich column-family data model, and causal consistency. EIGER is an evolution of COPS that targets large geo-replicated web services instead. Like COPS, EIGER tracks dependencies to ensure consistency, but instead of tracking dependencies on versions of keys, EIGER tracks dependencies on operations. It also provides a novel non-blocking read-only transaction algorithm that is partition tolerant and a novel write-only transaction algorithm that atomically writes a set of keys. Additionally, EIGER is lock-free, providing a lower latency, and does not block concurrent read transactions.

ChainReaction

ChainReaction [3] is a distributed key-value store that offers high-performance, scalability, and high-availability that can be deployed either on a single datacenter or on Geo-replicated scenarios over multiple datacenters. This solution offers causal+ consistency, globally and within each datacenter, and is able to leverage on the existence of multiple replicas to distribute the load of read requests. Similarly to COPS, ChainReaction also provides a transaction algorithm, called GET-TRANSACTION, that allows to get a consistent view (i.e, read operations) over a set of objects. Additionally, a stabilization algorithm was implemented to deal more efficiently with the metadata information that encodes causal dependencies between operations, allowing to preserve causal guarantees while keeping the metadata overhead low.

PNUTS

PNUTS [24] is a parallel and geographically distributed database system that was developed to support Yahoo!'s web applications. PNUTS provides a data storage organized as hashed or ordered tables of records with attributes, low latency for large numbers of concurrent requests including updates and queries, and novel per-record timeline consistency guarantees: all replicas of a given record apply all updates to the record in the same order. To keep response times low, PNUTS performs all high-latency operations asynchronous, that are carried out over a topic-based pub/sub system called Yahoo! Messsage Broker (YMB). Additionally, PNUTS is a hosted, and centrally managed database service that is shared by multiple applications, that leverages on automated load-balancing and failover to decrease operational maintenance complexity.

Spanner

Developed at Google, Spanner [25] is a highly scalable, globally distributed database that automatically splits data across Paxos state machines in response to changes in the the amount of data or number of servers. Additionally, it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner implements the strong consistency model with linearizability, and provides two important features: *i)* the data replication configurations can be dynamically controlled at a fine grain by the applications, allowing them to control read and write latency, durability, availability, and read performance; and *ii)* externally consistent reads and writes, and globally consistent reads across the database at a timestamp, which allow Spanner to support consistent backups, consistent MapReduce executions, and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These last features are enabled by the fact that Spanner assigns globally meaningful commit timestamps plus some uncertainty to operations, with the support of the TrueTime API. These timestamps reflect a serialization order that satisfies linearizability. The guarantees on Spanner's timestamps depend on the bounds of the uncertainty, that usually are small as GPS and atomic clocks are used for better precision.

2.6.3.2 Caching

Caching is a different approach to replication where the data is simply copied to a server, closer to the end user, in order to reduce read operations response time.

Tradionally, caching works with dedicated servers (for instance the Akamai platform [1]) although there are efforts to expand these infrastructures to the client side, such as in the case of Akamai NetSession [78] where the service can run on the client machine that connects to the Akamai infrastructure and extends the Content Delivery Networks (CDNs) to the clients. The client itself caches content, and then the clients share that content with each other in a peer-to-peer fashion. However it's the NetSession that decides where the data goes and what data is replicated by each client. It should be noted that in this case the replicated data is immutable. There are recent works, such as Legion [49], that allows to replicate portions of the state of a web application (data objects) in the form of CRDTs [45] to web clients browser, allowing mutations (i.e, state-modifying operations), that are done locally and then propagated between clients and eventually replicated to the server.

2.6.4 Discussion

Replication is an essential mechanism that will allow to elastically expand the discovery service out of data centers and onto the edge nodes. As edge nodes tend to have variable and possibly limited resources it is more advantageous to use partial replication for data replication. Additionally, multi-master replication should be leveraged, in order to provide low user-perceived latency. Since we want to provide not only low latency, but also high availability, weak consistency solutions should be prioritized.

Since the discovery service needs to be a distributed system, the need for propagation of information in between the various interacting components is fundamental. Thus, in order to be able to efficiently replicate information we need to look at information dissemination strategies.

2.7 Information Dissemination

Information dissemination is the propagation of information to the intended recipients while satisfying certain requirements such as delay and reliability, although these requirements can vary, depending on the information that is being disseminated [20].

Once data has been acquired locally by the service discovery, a decision has to be made about if and where to send the information about the registered instances. To ensure scalability and efficiency, service discovery replicas should rely on localized decisions, meaning that the acquired registrations should only be used by, and propagated to, components localized on the same node or in very close proximity (one wants to avoid having a master component that sees everything, knows everything, and takes all the decisions as this would be a bottleneck and a single point of failure). Ideally, we want that each area of an application deployment to have the freedom to (re)configure itself, which implies that the data needs to be shared locally, and organized in a hierarchic way.

The most used approaches for information dissemination are Gossip and Publish/Subscribe.

Gossip

Gossip-based dissemination tries to mimic the way rumors and epidemics spread over a population [28] by having all the participants collaborate equally to disseminate information among participants in the system, where each participant forwards messages received for the first time to a subset of the remaining participants [44]. In practice gossip is done over overlay networks. An overlay network is defined as a network which is deployed on top of another network, whose links are independent of the underlying network links and topology [44]. Overlays, in their essence, encode neighboring relationships among peers that are participating and collaborating in a given distributed protocol or system, where each peer manages a set with the information of its local neighbours.

In gossip there are three different approaches to disseminate information (messages): *i)* eager-push, where participants send the message to their gossip targets as soon as they receive it for the first time; *ii)* pull, where participants periodically query another randomly selected participant for information regarding recently received, or available, messages. When they become aware of a message that they did not received yet, they explicitly request that message to be sent to the neighbour that has it; and *iii)* lazy-push, where when a participant receives a message for the first time, it forwards only the message identifier to its peers. If a participant receives an identifier of a message they did not receive yet, they will explicitly request the message to be sent to them, similarly to the pull approach.

Relevant examples of this type of dissemination are MON [48], GoCast [72], and Plumtree [42]. We do not discuss these approaches in detail because they fundamentally use the techniques described above, in different combinations.

Publish/Subscribe

Publish/subscribe, as already explained in Section 2.4.1, is a messaging pattern that allows publishers to send categorized messages to subscribers interested in that topic, and in turn to disseminate information.

2.7.1 Discussion

As stated above, the discovery service should rely on localized decisions, disseminate information only to peers localized in very close proximity and be organized in a hierarchical way. This notion of locality between peers can be achieved with the help of gossip-based dissemination, as gossip protocols have in consideration neighbourhood relations between the participants in the protocol. Additionally, gossip protocols are highly resilient, since they have an intrinsic level of redundancy that allows them to mask node and network failures, scalable, as they distribute the load among all nodes in the system, low cost and simple to implement without the need of additional system components (as a broker in publish/subscribe protocols for example) [43].

2.8 Summary

In this Chapter we discussed techniques of cloud and edge computing, monolithic versus microservices architectures, forms of communication between microservices and current relevant solutions, replication of data and application logic, and information dissemination. All of the abstractions offered by these different areas will be important for the realization of the work described in the next Chapter, as our solution will take advantage of some of the discussed solutions, either by directly using them or adapting them to deal with concrete problems, which we will discuss in the next Chapter.

Chapter 3

Sheik: Design & Implementation

In the previous chapter, we discussed existing solutions that support service discovery however, we have shown that none of the existing solutions is adequate to be deployed on the edge, as they are too computationally and storage heavy.

In this chapter we propose a novel proposal, named Sheik, that is a distributed registration and discovery service that allows to dynamically bind and locate microservices in cloud/edge settings. We start by defining our system model (Section 3.1) through a set of assumptions, that motivate our design choices (Section 3.2). We then present a high level overview of the Sheik's architecture (Section 3.3), and discuss its main components. Finally, we detail how we have implemented the service using the Java language (Section 3.4).

3.1 System Model

Sheik is a microservice discovery service tailored for hybrid cloud and edge ecosystems, as such the (network) location where each instance executes must be very well know and those locations should be highly available. Thus, we assume that Sheik replicas will be running in datacenters (taking advantage of cloud's elasticity, geo-distribution and ubiquitous network access), and potentially some replicas could be deployed outside of the data centers and closer to the clients, taking advantage of edge devices (with enough computational resources and stability).

We also assume a mechanism to protect the access to the service is already in place. Typical access control policies employed for services in the cloud should suffice. Hence, we do not deal with this aspect of the system design in the scope of this thesis. Finally, we assume that replicas that may fail are not byzantine, that is, Sheik replicas or microservice instances can fail but do not deviate from their prescribed behaviour, and once they fail/crash they stop all communication.

3.2 Design Choices

Decentralization

One of the requirements for Sheik to be able deal with the highly dynamic edge environments, and provide high availability and scalability is to be replicated. Thus, instead of having one Sheik replica that stores all the data and deals with all the requests (centralized system model), there are multiple Sheik replicas, each of which stores a fraction of the data and deals with a part of the requests (i.e., it resorts to partial replication). There isn't a single point where all the decisions are made, but rather every replica makes a decision for its behaviour based on its knowledge of the state of the system.

Following this system model, allows to provide fault-tolerance and high availability, since even if some of the Sheik instances fail or become unreachable, microservices can still interact with the system and keep their normal operation.

Hierarchy

Sheik replicas self-organize into a tree structured hierarchy, where each Sheik represents a node in the tree. This hierarchy is maintained with the help of location tags as exemplified in Figure 3.1. Location tags all start with the word *global* (a virtual tag whose purpose is to indicate the root location level) and are a sequence of location names separated by a point ("."). Location tags start small, typically representing a more broad location such as continents or countries (*global.eu* for example), and get bigger and with additional levels further away from the root the instances of Sheik are, thus closer to locations with clients (*global.eu.pt.lx* for example). When a new Sheik is deployed, the idea is that he will be responsible for a more specific area than the Sheik above him.

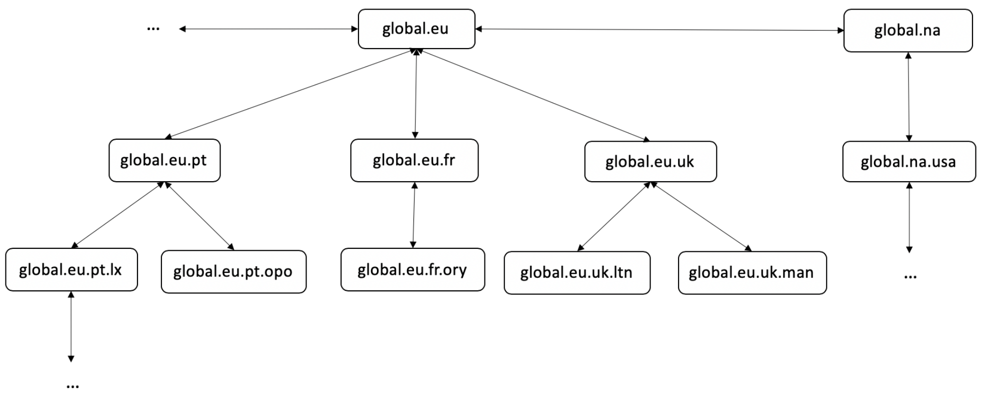

Figure 3.1: Example of a Sheik hierarchy

Locality

Ensuring locality is one of the main Sheik goals, and it has a big impact in the way data (regarding different instances of microservices) is replicated and how the different instances of Sheik interact with each other. To provide efficiency in response times, Sheik replicas rely on localized decisions and propagate data only to other Sheik replicas in close proximity.

This leads Sheik to, when queried for instances, attempt to return the closest instance that is available in his registry (although it provides other querying alternatives that can be exploited by applications with special needs), and instances are consistently being redirected to communicate with the closest Sheik.

This locality is ensured with the help of the location tags discussed above, where all instances (whether it is Sheik instances or microservice instances) have a location tag associated to them. Instances that have tags with a larger common prefix are presumed to be closer, regardless if it's a pair of Sheik instances, microservices instances or a Sheik instance and a microservice instance. For example, tags *global.eu.pt.lx* and *global.eu.pt* have a larger common prefix and thus are considered closer, than tags *global.eu.fr* and *global.eu.pt.lx*.

Partial Replication

As stated above, ensuring locality has had a big impact in the mechanics that we use for Sheik to replicate information about its registered microservice instances or about other Sheik replicas. Since most of the times we want to return the closest instances of a service to answer a query there is no need for all Sheik replicas to keep local registry concerning all the registered instances in the system, but only of the ones that are closest. In order to achieve this, the concept of neighbours is used, where a Sheik replica is considered neighbour of another if its immediately above or below it in the location tag hierarchy (tree), where we consider all root replicas are neighbours of each other. This idea is exemplified in Figure 3.2, where replica *global.eu.fr*, for example, has as neighbours replicas *global.eu* and *global.eu.fr.ory*. The information is then replicated in a two hop fashion: *i)* the first hop is when a Sheik replica propagates a message of a state changing operation to its (direct) neighbours (this operation can be the update of a registry for a microservice) *ii)* the second hop is when a Sheik propagates a message containing some state update received from a neighbour to all of its neighbours. The dissemination stops when the origin of the message received is from a Sheik that is not part of its (direct) neighbours, or the message didn't lead to any changes in the state for instance, because the information contained within the message was already known by the receiving Sheik.

This type of partial replication not only allows to ensure locality but also to provide availability and fault tolerance, as microservice instances can always failover to a neighbour of the Sheik instance they were connected to.

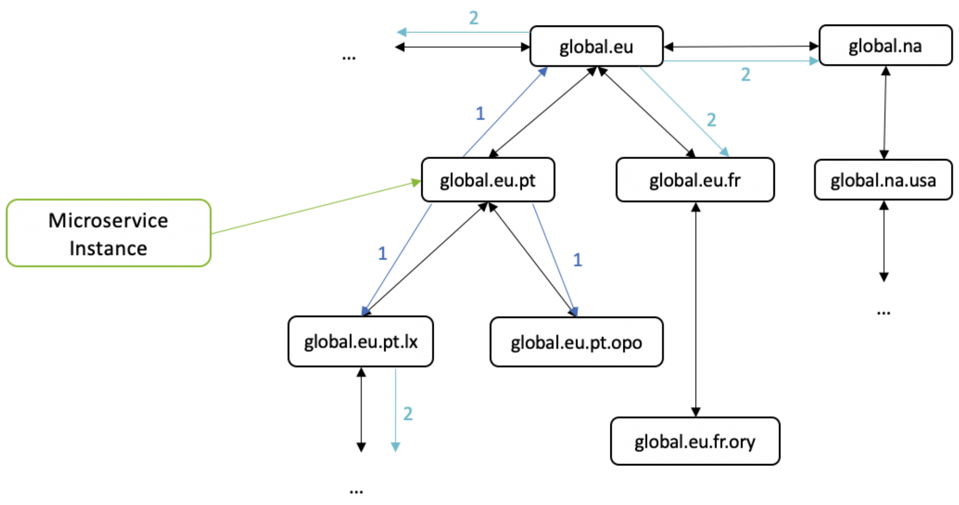

Figure 3.2: Example of a partial replication of a registry of a new microservice instance.

Client Side Caching

In order to reduce the amount of requests to the server and reduce response times perceived by the clients, information about microservice instances is cached for a period of time on the client side. This allows the client to retrieve the information needed about its most used instances faster and to considerably reduce the load imposed on Sheik.

Client side caching is also relevant in case of a Sheik failure, as instances can keep using the cached information whilst they failover to another Sheik, making the failover between Sheik instances more seamless to clients.

3.3 Architecture

When developing Sheik, we started from some insights introduced by Eureka [57]. Much like Eureka, Sheik is also a registration and discovery service, targeted primarily for microservice instances, where a dedicated server is responsible for maintaining an updated registry of all deployed instances. Additionally, we also decided to separate Sheik into two components: Sheik Server and Sheik Client.

The core Sheik mechanic is maintaining a registry of available microservice instances in order for other microservices to be able to discover and communicate with them. To this end, a microservice needs to register itself with the service and provide metadata such as an IP address, port, service name, and location. Additionally, during registry a microservice instance may provide any additional metadata it considers useful, and later update it whenever necessary. Sheik does not interpret or act on that metadata but exposes it to clients that send queries regarding that microservice or microservice instance. In order to certify the availability of microservice instances, Sheiks expects a heartbeat message from each registered instance, that should be sent at regular intervals.

If an instance fails to send three consecutive heartbeat messages, it will be considered as failed and removed from the registry, but an instance can also gracefully unregister itself from the service when is terminated, by using operations exposed in Sheik API for that purpose.

Since Sheik is replicated, it also maintains a registry of Sheik instances in order to be able to disseminate information among its peers. Similarly as with microservice instances, when a new Sheik instance is deployed it registers itself with another known peer and needs to provide metadata such as an IP address, port, and location tag. A regular heartbeat message is also expected from all registered Sheik instances, and if an instance begins to consistently fail to send a heartbeat it will be considered failed and removed from the registry. At regular intervals, Sheik sends a configurable amount of messages with state updates to its neighbours, and uses this messages to disseminate information regarding existing microservices instances at a two-hop distance as explained before.

All these mechanics are divided between the two Sheik components, and some are executed in the Sheik Server component while others depend on the Sheik Client component, with some mechanics being implemented by having the two components cooperate. Sheik Server is the discovery server and is responsible for maintaining the registry with the information about available microservice and Sheik instances, and for disseminating this information among its neighbours. Sheik Client is a smaller Java-based client component that abstracts all the interactions with the server from the client through a simple and easy to use set of operations, and in this way turns the interactions between Sheik and microservices much easier. Additionally, in the Sheik Client component are implemented the failover and caching mechanisms, which avoids microservice developers to be concerned with this aspects.

Sheiks architecture is illustrated in Figure 3.3, where we can see how these two components interact with each other and with the microservice instances.

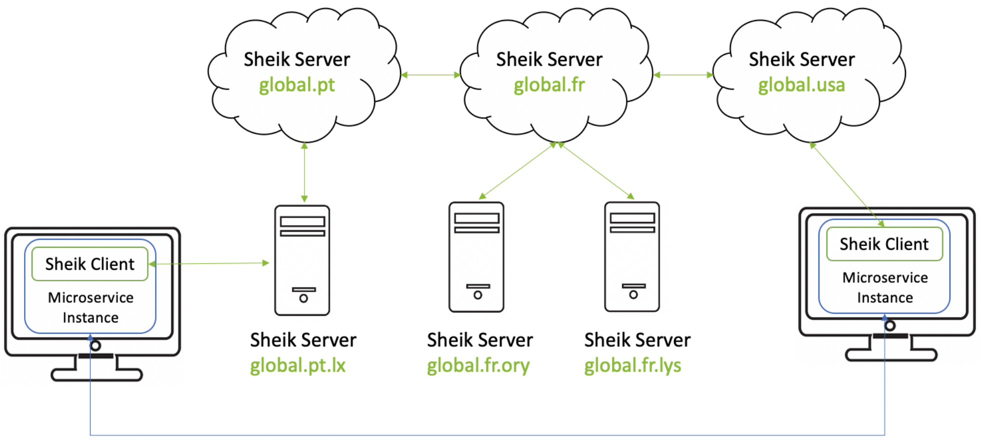

Figure 3.3: Simplified Sheik's Architecture. The green lines represent represent communication between Sheik replicas, and the blue lines represent communication between microservice instances.

In order for microservices to be able to communicate with the Sheik Server, Sheik Client provides a set of well defined and easy to use operations:

register Registers the microservice in the system, and then periodically sends a heartbeat.

shutdown Gracefully unregisters the microservice from the system.

update_metadata Allows the microservice to update its metadata, either by providing new information or updating an existing one.

getOne Retrieves one random microservice instance with a given application name.

getClosest Retrieves the closest microservice instance with a given application name.

getAll Retrieves all microservice instances with a given application name.

3.4 Implementation Details

Sheik's prototype was developed in the Java programming language on top of Apache Tomcat [8] and includes all of the functionalities described above. Additionally, Sheik uses leveldb [46], a lightweight open source key-value storage library written at Google, to store its data. leveldb provides an ordered mapping from string keys to string values, that are stored on disk as byte arrays.

We now give some details on Sheik's implementation and main features.

Configuration files

As a starting point, Sheik comes with two configuration files, one for each component, that need to be set up before start up.

In the Sheik Server configuration file, called *sheik-config.properties*, the user needs to set up the port, the IP address, the location tag and the URL of a peer to register with (or several URLs separated by a space). If the Sheik Server is running in standalone mode or is the first replica, and thus has no peers yet, this property needs to be set up to a special keyword *NONE*, indicating that it has no other instance that it needs to register with. Additionally, the user can set up the number of retries to send a message in case of failure and the number of messages to be sent in a batch when replicating information.

In the Sheik Client configuration file, called *client-config.properties*, the user needs to set up the port, the IP address, the location tag, the application name, and the URL of a Sheik Server to register with (or several URLs separated by a space). Additionally, the user can define some of the cache's behaviour, namely, the user can set up the cache size, the expiry period after which the elements in cache are no longer valid, and the interval for the cleanup function. Finally, the user can also set up the number of retries to send a message in case of failure.

Communication Mechanism

By default, the Sheik Client uses the Jersey Rest client library and GSON along with JSON payload to communicate with the Sheik Server, that in turn exposes a RESTful API. Similarly, the Sheik Server also uses Jersey and GSON to communicate with its peers. For this, the Sheik Server exposes two distinct endpoints: */peer* for communication between Sheik Servers, and */sheik* for communication between the Sheik Client and Sheik Server. Notice that the Sheik Client is executed by microservices that register and use Sheik to identify other microservices that they need to interact with.

Since the Sheik Server exposes REST based endpoints, microservice instances can make their own HTTP requests without resorting to the Sheik Client component. This is useful for services that are non-Java based, as in this way they can still take advantage of the discovery service, but will need to implement the requests in the language of the service. Additionally, the Sheik Client can easily be implemented in other languages. To do so however, falls outside of the scope of this thesis.

Registration Protocol

Both Sheik components need to register themselves with the discovery service at start up. For this, both components follow a very similar registration protocol.

First, the URL (or a list of URLs) of the discovery service is retrieved from the configuration file and a message with all the necessary information is created. This message contains the IP address, port, location tag, and, in case of microservice registration, the application name and instance id. The instance id is a combination of the instance's location tag and a universal unique identifier (UUID), such as "global.eu.pt.4099b700-e786-11ea-adc1-0242ac120002" for example, and was designed in this way to be able to take advantage of the information about an instance's location directly from the id while still have unique identifiers.

Then a registration request is sent to the first Sheik URL in the list. If the request is successful, the information is processed: *i)* in case of microservice registration, the location tag of the discovery service is returned, that is stored and then later used by the failover protocol to find closer Sheik Server instances; *ii)* in case of a Sheik Server registry, the discovery service returns all of Sheik and microservice instances that it currently contains in its registry that is then stored, providing to the newly joined instance up to date information regarding the current system state.

If the request fails, it is retried a few more times, waiting a few seconds in between each attempt. If it still fails, the protocol tries the next URL in the list, until one of them is successful or there is no more URLs in the list. When the entire list is scrolled through, and there is still no success, the protocol waits for 30 seconds before trying again, starting at the top of the list. This interval is used to allow intermittent connectivity or network issues to be overcome, and the situation to normalize, as there may be network partitions or the discovery service might be temporary down for some reason.

Only when the registration is successfully completed, will the components proceed with all the other mechanisms and allow requests to be issued.

It is important to note that, in case of a Sheik Server registry, if instead of an URL it is retrieved the keyword *NONE*, the registration protocol does nothing as it means that this instance is either running in standalone mode or has no peers to register with yet.

Additionally, it is recommended that the URLs provided to microservice instances to register with are the closest possible (although there is a mechanism that tries to redirect the microservice to the closest Sheik Server instance when the registering replica identifies this scenario, by taking into account the location tags), and that Sheik Server instances register themselves with the instance directly above them according to the location tags hierarchy.

From the server side, when a Sheik Server receives a registry request it first verifies if all the required information is provided. In case some of the information is missing the request fails, otherwise the server proceeds to verify if the instance already exists in the system, and here the procedure changes depending on what type of instance is being registered.

In case of microservice instances, the instance ID provided should be unique. If there is no other instances with the same ID, the instance is registered in the system, the instance's origin tag is set to the server's location tag, a new message is added to the queue (to disseminate this new information) and the location tag of the server is returned. Otherwise, if there is already an instance with the same ID, the server verifies if the new instance's origin tag is the same as of the registered instance. When a microservice instance first registers with a server an origin tag is assigned to them, that is the same as the servers location tag. This origin tag is used as a control tag and, if the origin tag is the same, this means a duplicate registry and the request fails. On the other hand, if the tags are different this means a failover registry (the original Sheik Server crashed or this instance is closer) so the origin tag of the registered instance is changed to the new tag (the location tag of the Sheik Server where the instance is registering) and a new message is added to the queue to disseminate this change.

In case of another Sheik Server instance, the location tag should be unique and if there is already a instance with the same location tag the server just returns its current registered Sheik and microservice instances, because the registering Sheik Server instance may have crashed and is starting up again faster than the eviction protocol could handle the failure. Otherwise, the new instance is added to the registry, the neighbours list is recalculated, a new message is added to the queue to disseminate this change to the neighbours and a list with the current registered Sheik and microservice instances is returned.

Renewal Protocol

After the successful registration, instances need to send heartbeats at regular intervals to indicate that they are still alive. For this, a new thread is created that sends heartbeats every 30 seconds. In case of microservice instances, heartbeats are only sent to the Sheik Server with which they registered and the heartbeat message needs to include the application name and instance ID. In case of Sheik Server instances, heartbeats are sent to all neighbouring instances of Sheik and the heartbeat message needs to include the location tag (since there can only be one Sheik Server per location tag).

At the server side, when Sheik receives a heartbeat message, it first verifies if the instance exists in its registry. If it does, it then verifies if the timestamp of the received heartbeat is more recent than the one it has registered. If it is, the new timestamp is registered and a new message is added to the queue to disseminate this change to the neighbours as prescribed by the dissemination strategy employed between Sheik instances. If any of the previous conditions are false, nothing happens.

Information Dissemination Protocol

In order to disseminate information among its peers, Sheik uses a gossip based protocol, where all operations that are performed on the server, that modify its state, are replicated to all of the neighbour nodes with a limited horizon of two hops [44].

Whenever Sheik receives a request such as a registry, a heartbeat or a metadata update, considered a state modifying operation, a message detailing the operation is created, containing information such as the origin and the sender of the message (which in this case are the same, and equal, to the server's location tag), the type of the operation and the information needed to perform that operation, are added to the queue to be disseminated to neighbouring Sheik instances. Because a particular neighbour can be temporarily down or overloaded, causing it to be unable to receive updates, there is a messages queue for each neighbour. Then, Sheik periodically sends batches of messages to each neighbour, starting with the oldest and iterating over neighbours. If the message is sent successfully it is removed from the queue, if on the other hand, the request fails, a special counter is incremented counting the number of attempts to send the message, and at the third retry the message is removed. This set of actions establishes the first hop of the information dissemination protocol.

Whenever Sheik receives an update request from another Sheik, it tries to execute the operation in the message. If the update produces a change in the state, it then becomes a candidate for further dissemination. If the update didn't produce any changes in the state, it will no longer be propagated. If the instance trying to be updated doesn't exist in the registry (this excludes register operations), Sheik will ask the sender of the message for the full instance information and store it, and in this way recover the missing information about that instance. When a message becomes candidate for dissemination, Sheik verifies if the origin of the message is one of its direct (i.e., one hop) neighbours. If it is, the sender information is updated and the message is added to the queue of all neighbours excluding the origin and the sender neighbours. On the other hand, if the origin is not a neighbour, the propagation of that message stops. This set of actions establishes the second hop of the information dissemination protocol.

It is important to note that, as a way to reduce the amount of messages sent between Sheiks, heartbeat messages sent by microservice instances are not replicated. The server with which the microservice instance registers is responsible for keeping the heartbeats valid and notify its peers when the instance fails, that in turn will assume that those instances are alive until they receive a shutdown notice or the Sheik responsible for those instances fails.

Eviction Protocol

When an instance fails to send three consecutive heartbeats it is safe to assume that the instance may have failed. In order to remove dead instances from the registry Sheik employs an eviction protocol, that periodically runs through the registry and verifies if the instances have valid heartbeat timestamps. When the heartbeat has expired the instance is removed from the registry. In case the expired instance is a Sheik Server instance, besides removing the instance from the registry, all of the messages addressed to this instance and all of the microservice instances that have the same origin tag as the instance location tag are also removed from the registry. It is important to note, that this microservice instances will not be lost as the failover protocol will deal with this problem, and if the microservice instance still got removed it is because it wasn't able to failover and thus its most likely crashed.

Since microservice instances heartbeats are not replicated, the eviction protocol only verifies the validity of the timestamps of instances that registered directly with the server, meaning that they have the same origin tag as the server's location tag, and will assume all the other instances are alive until told otherwise. It is the responsibility of the server to communicate to its peers that an instance is no longer valid, and this is why, when a peer is considered failed, all of the instances registered with that server are removed from the registry, as that server will no longer able to notify others of dead instances.

Failover Protocol

In order to provide more resilience to the service, a failover protocol is implemented in the Sheik Client component. The failover protocol is responsible for switching to another Sheik Server when the current upstream Sheik server with which the client is communicating fails, and, if it finds an instance that is closer that its current upstream node, it will switch to that server without the current one failing.

For this, the client periodically fetches a list of Sheiks from the server, and verifies if any of the new Sheiks is closer than the current. If it is, communications are switched to the new server. At any point, if the client is unable to make three consecutive requests to the server, it switches to another Sheik from the list (giving preference to the closer ones).

Since when failing over, a candidate Sheik is searched from the full registry and not just from the neighbours, there is a chance that the new Sheik doesn't know about the instance and there is also the need to change the origin tag to the new Sheik. Thus, when the instance failsover it re-registers itself with the new Sheik.

In order to request a list of Sheiks to failover the client needs to provide its location tag. Then, Sheik tries to find at least two Sheik instances to return following the following order: *i)* first, if the current Sheik is a top instance (meaning it only has two tags: *global* plus another one) then it needs to return two brother instances; *ii)* if the Sheik tag is the same as the provided tag, then it searches two instances above; *iii)* if the tag are different, it tries to find a Sheik that has the same tag as the provided one, and fills the rest with instances above.

Caching

In the Sheik Client component a cache is also implemented. Having a cache allows to store locally the instances retrieved from the server, so that future requests for that data are served faster than by constantly accessing them from the server, allowing to efficiently reuse previously retrieved data. Maintaining data validity is very important, thus control such as TTLs (Time To Live) are applied to expire the data accordingly.

In this cache we have two different TTL control parameters. Since the cache is implemented as a LRUMap, there is a timestamp for each key in the map, where each key is the application name and the value is a custom Java object that stores the results of different queries (Sheik offers three types of requests to obtain microservices information: get one random instance with given application name, get the closest instance with given application name, or get all registered instances with the given application name). Whenever a key is accessed, its timestamp is updated. Later, a periodic function runs through the map entries and eliminates the expired ones (applications that are the least used), thus saving the most frequently used entries but still making room for new ones. The second TTL control is on the results returned by the requests. For each type of request, a timestamp is saved from when the information was acquired, and not updated by reads. When the client requests data, Sheik Client first verifies if the data is available on cache and if the timestamp is valid. If is, cached data is returned, on the other hand, Sheik Client requests the data to the server, stores it in cache (either creates a new entry, or overrides the existing one updating the timestamp) and then returns it to the client.

It is important to note that, since the LRUMap deletes the oldest entry when full, we do not worry with that part when adding new entries to the map.

3.5 Summary

In this Chapter we presented the first contribution of this thesis, the Sheik service discovery service. We began by defining the system model and motivating the design choices behind Sheik´s implementation, we then detailed Sheik's architecture and its provided implementation details.

In the next Chapter we present Sheik's comparative experimental evaluation with Eureka, a relevant state-of-the-art solution.

Chapter 4

Evaluation

In this Chapter we present our experimental work. At a higher level, we aim at demonstrating the applicability of our solution in realistic scenarios. At a more detailed level, we aim to find answers to three vital questions: whether our solution introduces any overhead in the application operation, how our solution scales and in what scenarios is better when compared to another relevant state-of-the-art solution (Eureka). To this end, we begin by describing our use case application (Section 4.1) and experimental methodology (Section 4.2), following by presenting the tools that we used to help us in conducting our experiments (Section 4.3), and finally detail the experimental evaluation of Sheik where we compare it with Eureka [57], a relevant state-of-the-art solution that serves as a performance baseline to our own solution (Section 4.4).

4.1 Use Case Application

As our solution is a microservice discovery service there was the need of an use case application that used the microservice pattern that could interact with our solution. Current open source solutions are too bulky and introduce a lot of noise, as they already provide some sort of message dissemination mechanism between the microservices, have an API gateway, and are connected to some sort of database, as is the case of eShopOnContainers [32]. Thus, we developed our own simple use case application with six microservices (cart microservice, catalogue microservice, orders microservice, payment microservice, shipping microservice and user microservice) whose arquitecture and interactions are depicted in Figure 4.1. This application was inspired by the eShopOnContainers application referred above.

This use case application mimics a basic online shop, with each of the microservices being responsible for one specific task: the *catalogue* miscroservice is responsible with loading the pages with the items to sell and storing items information; the *cart* microservice is responsible for managing the user's cart; the *orders* microservice is responsible for processing the orders; the *payment* microservice is responsible for processing the payments; the *shipping* microservice deals with processing the shipping; and finally, the *user* microservice deals with managing the user's accounts. As the purpose of this use case application is to aid in the experimental evaluation, we are not interested in fully developing the application but just in exposing an API through witch the microservices can communicate with each other and simulate active work when handling direct requests. Additionally, some of the microservices were purposely designed in a way that they need to communicate with other microservices to be able to finish client requests, such is the case for: *i)* the cart service, that to be able to add items to the cart, first requests the items description from the catalogue service; *ii)* the orders service, that to be able to confirm an order needs to contact the cart service (to obtain the list of items in the user cart), the payment service (that should process the payment on an external service), and the shipping service; and *iii)* the shipping and the payment services each need to contact the user service to obtain user information to be able to process shipping and billing, respectively. It is important to note that this is just the logic behind the design of the application, in reality the implementation of the microservices is just an API that makes requests to the discovery service to get the address of the relevant microservices that have to be contacted and executes requests over these microservices, using a think time (i.e., sleeping) of 5 milliseconds (15 milliseconds in the case of the payment microservice, as usually it needs to contact an external payment service and this kind of secure processing takes additional time). The think time emulated the time spent processing the request including IO operations, logging, or accessing databases.

4.2 Experimental Methodology

Throughout our experimental evaluation we intend to demonstrate the applicability of our solution and to understand what overhead, if any, is introduced on microserservices performance when using our solution, in what scenarios our solution is better than Eureka, and how scalable our solution is. To this end, we resort to implementations of a prototype of our solution and Eureka, and execute them in a geo-replicated cluster, as a way to simulate multiple datacenters or possibly one datacenter and multiple edge devices in one location, with the help of our use case application.

In order to be able to find answers to our questions and evaluate the performance of our solution, we consider eight different deployment scenarios (Figure 4.2):

Scenario 1 The first scenario is the simplest one without any replication, it involves one discovery service replica and one replica of our use case application all deployed in one location;

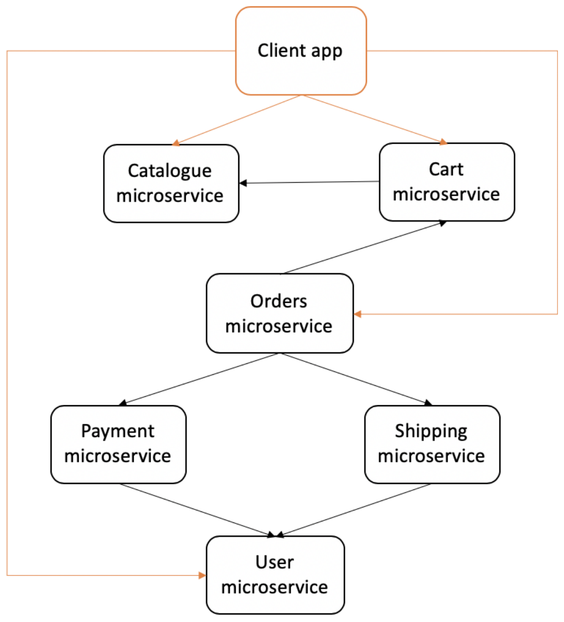

Figure 4.1: Use Case Application Architecture

Scenario 2 The second scenario introduces replication of our use case application, and it involves one discovery service replica in one location and three replicas of the use case application spread across three locations;

Scenario 3 In the third scenario both the discovery service and the use case application are replicated, thus in this scenario there are three discovery service replicas and three use case application replicas spread across three locations;

Scenario 4 The forth scenario introduces node failures in our experiments, resorting to a setup similar to the one used in scenario three, where after two minutes of execution one of the discovery services fails, forcing the microservice instances to failover to another discovery service;

Scenario 5 The fifth scenario also introduces node failures in the system, being similar to scenario three, but this time two of the microservice instances fail (the first one fails after one minute of execution, and the second one after two minutes, from the start of the experiment).

Scenario 6 The sixth scenario tests the performance of the solution in a situation where the use case application is only partially replicated across the locations. Thus there are three discovery service replicas, one in each of the three locations, and there are only three microservice instances in each of the locations. As there are nine microservices in total, some of them are replicated, but there is at least one copy of each microservice, in order for the use case application to function as intended;

Scenario 7 The seventh scenario is similar to the forth, where the discovery service fails after two minutes of execution, but this time the system is deployed using an initial configuration similar to that of scenario six;

Scenario 8 The eight and last scenario is similar to the fifth scenario, where two of the microservice instances fail after one and two minutes from the start of the experience, but this time the system initial configuration is similar to that of scenario six.

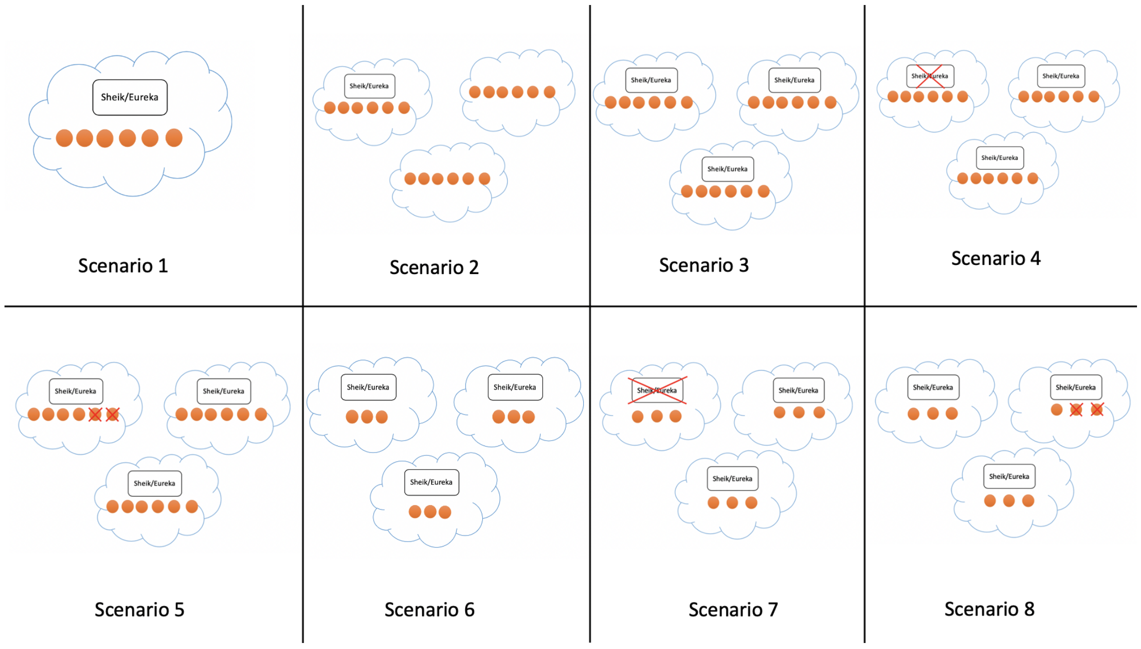

Figure 4.2: Schematization of the eight deployment scenarios

4.3 Experimental Tools & Setup

As mentioned above, the experiments where executed with the help of Grid'5000 [15], a large-scale and flexible testbed for experiment-driven research spread across eight locations with 37 clusters of which we used four: grimoire and grisou in Nancy, parasilo in Rennes, and chetemi in Lille. We used these four clusters in particular because they are relatively at an equidistant location from each other and have similar characteristics: each machine in the grimoire and grisou clusters have two Intel Xeon E5-2630 v3 processors with eight cores/CPU and a network speed of four times 10 Gbps plus 56 Gbps InfiniBand, the parasilo cluster has machines with two Intel Xeon E5-2630 v3 processors with eight cores/CPU and a network speed of two times 10 Gbps, and the chetemi cluster has two Intel Xeon E5-2630 v4 processors with 10 cores/CPU and a network speed of two times 10 Gbps. Initially, grimoire was used in combination with parasilo and chetemi to retrieve the overall performance results. But when we performed the scalability tests, there was a need to change grimoire to grisou as grisou had a significantly higher number of nodes in the cluster, allowing to deploy more JMeter instances.

In order to emulate a high number of clients, we used JMeter [5] that allows to perform load and performance tests on static and dynamic applications, by simulating a heavy load on the servers, and to obtain metrics such as latency and throughput. JMeter emulates multiple clients by executing multiple threads that interact with the service.

When running the experiments, we separated the different components on different nodes whitin the cluster: the discovery service replica is running on one node, the use case application is running on another node, and the JMeter tests are running on different nodes from the previous ones.

We now detail our experimental work to evaluate our proposal using previously described methodology and tools.

4.4 Experimental Evaluation

In this Section we report our experimental evaluation of Sheik, that is divided into three parts. The first part provides insight into Sheik's overall performance and overhead, when compared to Eureka. The second part provides a more detailed study of Sheik's performance by analyzing its performance over time, when compared against the baseline solution. In the third part, we perform a scalability study, in order to understand how our solution is able to handle a growing amount of work by adding more concurrent requests to the system.

4.4.1 Overall Performance and Overhead

To evaluate Sheik's and Eureka's overall performance and overhead we resorted to eight different scenarios of deployment described in Section 4.2. For each of the scenarios we performed three different tests using JMeter to generate the load and extract the performance metrics, where, for each test, we varied the duration and the number of clients: *i)* the first test consists of 20 threads running during five minutes, *ii)* the second test consists of 20 threads running during 10 minutes, and *iii)* the third test consists of 40 threads running during five minutes. Each thread makes four requests to simulate a normal interaction with an online store: one to the catalogue microservice (to simulate a request to display the items), two to the cart microservice (to simulate items being added to a cart), and one to the orders microservice (to simulate an order being placed). Each test was repeated three times, to obtain more accurate results, and from each test, we obtained metrics such as the throughput and latency, that will serve as performance metrics.

At a first glance, we want to understand the overall performance of our solution, thus, for each scenario, all the results were averaged to obtain an a high level view of Sheik's performance when compared to Eureka. Figure 4.3 reports our results and shows the average throughput (in the y-axis, in operations/s) obtained for each scenario (in the x-axis) for each Sheik and Eureka (represented in blue and grey, respectively). The results show that not only Sheik's performance is on pair with Eureka's, but it is also slightly better, displaying, on average, a 5% higher throughput, even reaching 33% in scenario 5. Although the throughput in the last three scenarios seems low, this is because in these scenarios the use case application is partially replicated across the different locations, which leads to longer communication times between microservices and an overall decrease in the number of operations performed.

Figure 4.4 reports the client perceived latency observed in the experiments and shows the average latency (in the y-axis, in milliseconds) obtained for each scenario (denoted in the x-axis) for Sheik and Eureka (represented in blue and grey, respectively). The results show that Sheik's overhead is negligible in all cases, as on average microservices take 17ms to process the requests (5 ms to get the catalogue, 10 ms to add an item, and 35 ms to place an order, excluding communication delays) and the average latency, in fully replicated scenarios, is just 18 ms. Following the same pattern, Sheik also demonstrates an overall lower latency than Eureka, where, on average, Sheik's latency is 5% lower than Eureka's. Similarly to the throughput, latency is higher in scenarios 6, 7, and 8, for the same reason.

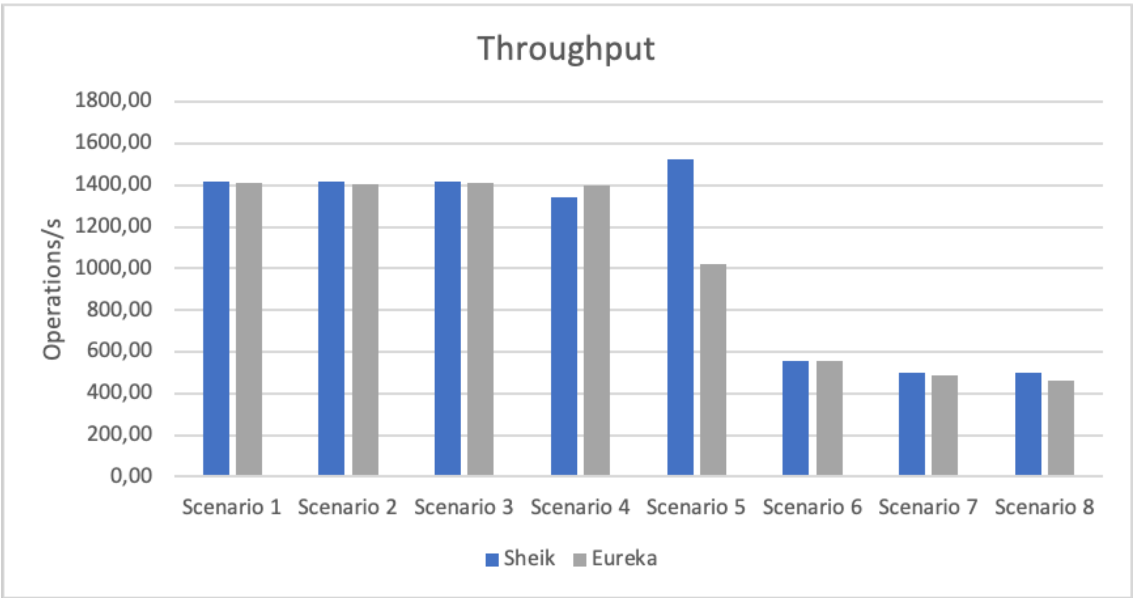

Figure 4.3: Overall Throughput

The most noticeable difference in performance between the two solutions is in Scenarios 4 and 5, where in Scenario 4 Sheik displays a 4% lower throughput and an equal 4% higher latency, and in Scenario 5 where Sheik displays a 33% higher throughput and a 31% lower latency, when compared to Eureka.

The slight difference in performance in Scenario 4 can be explained by the delay introduced by the failover protocol, as when an instance failovers to a new discovery

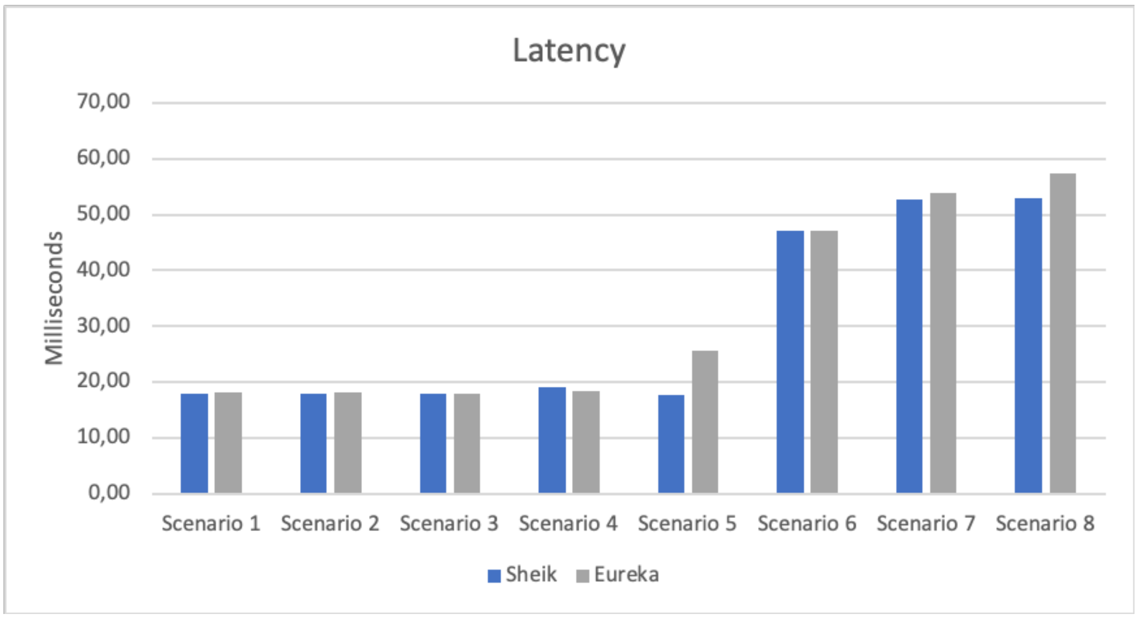

Figure 4.4: Overall Latency

service it needs to re-register itself with the new service. In Eureka this problem doesn't exist, because the Eureka state is totally replicated across all discovery service replicas, it makes no difference to which discovery service the instance is communicating with as long as the discovery service is in the same region as the instance. In Scenario 5, the higher observed values result from the requests being processed faster, since the microservice instances can't communicate with other target instances that were failed (for a short period, until the system stabilizes) the requests fail immediately, resulting in a temporary spike in throughput and a drop in latency.

Although this still happens is Scenarios 7 and 8, it's less noticeable as the throughput and latency are lower, the impact of the failure described above is not as noticeable.

We should note that, while the benefits from Sheik in terms of throughput and latency might seem modest, that this service is in the critical path of many interactions, and hence, the larger and more complex applications with hundreds or thousands different services, the benefits should be much more noticeable.

4.4.2 Detailed Study

In this Section, we are going to present in more detail what happens in each of the performed tests for each one of the eight scenarios.

First, we start by presenting the performance of both solutions obtained in the first test for each scenario (divided in three groups for better readability: the baseline group (scenarios 1 and 2), the fully replicated group (scenarios 3, 4, and 5), and the partially replicated group (scenarios 6, 7, and 8)). The objective behind this test is to obtain a comparative benchmark performance of both solutions. Figure 4.5 presents the throughput (in the y-axis, in operations per second) obtained in the first five minutes (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show a pattern with a high variation of values over time for both scenarios, with Sheik's performance being slightly above Eureka's. This is because some requests take longer to process than others (it takes 35 ms to place an order versus the 15ms to add an item to the basket, excluding communication delays). Thus, when there is a predominance in the heavier requests the throughput decreases, and when there is a predominance in the lighter requests the throughput increases, resulting in the zigzag lines.

Note that clients only issue their next operation after receiving the answer for the previous one (i.e., client requests are blocking).

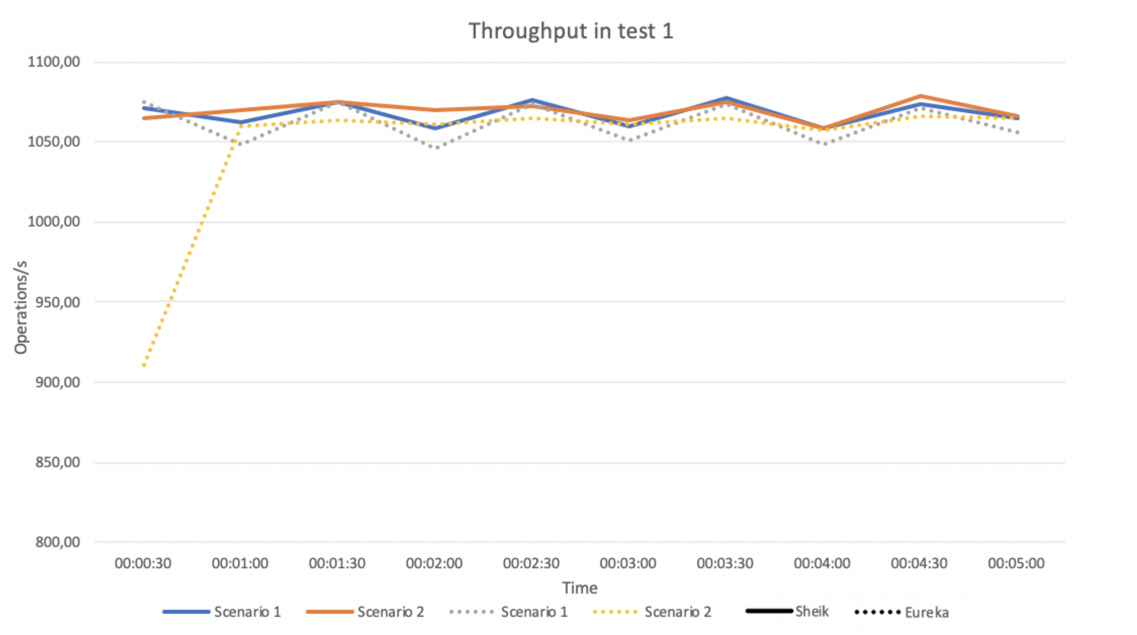

Figure 4.5: Throughput over time in scenarios 1 and 2 in test 1.

Figure 4.6 presents the throughput (in the y-axis, in operations per second) obtained in the scenarios 3, 4 and 5 during the five minutes of the experience (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show that, handling with replication introduces no noticeable overhead in Sheik's operation, as in scenario 3 the results are similar to those obtained in the non-replicated scenario 1. Additionally, since Sheik prioritizes closer instances, we see that the throughput is the same as in scenario 1, which means that microservice instances are indeed communicating with the closest instances on the same cluster although there are now additional instances in other locations. Analyzing the results of scenario 4, where the Sheik instance with which the microservices are communicating crashes on the second minute of the test, we can see the slight impact of the failover protocol, resulting in a temporary drop of performance while the client switches to another Sheik instance. Although, the system quickly recovers (in less than 30s) and is able to deliver the same performance as before. We can also confirm, that in the case of Eureka, the results remain the same as in scenario 3, as Eureka needs to do nothing as an Eureka replica fails because its data is fully replicated between replicas. In the fifth scenario, we can see that both solutions suffer a transient improvement in throughput when the instances fail (one instance fails on minute one, and another instance fails on minute two). This is because, as instances are unable to connect to the failed microservice instances in order to complete the requests, the requests fail immediately and thus are processed faster. As the situation normalizes, and the failed microservice instances are removed from the Sheik registry (note that the eviction protocol waits for three failed heartbeats in order to assume an instance is failed), we see a decrease in throughput as the communication is redirected to microservice instances that are more distant from the requester, which leads to longer communication times and a higher latency for handling client requests.

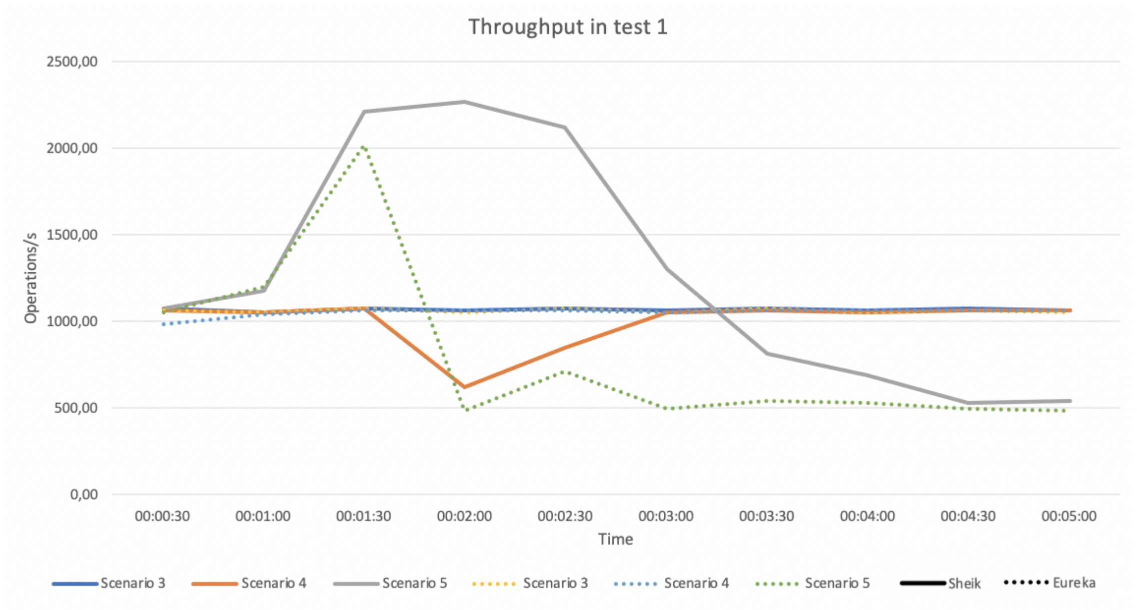

Figure 4.6: Throughput over time in scenarios 3, 4, and 5 in test 1.

Finally, Figure 4.7 presents the throughput (in the y-axis, in operations per second) obtained in the scenarios 6, 7 and 8 over the time of the experience (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show that Sheik is very consistent and is able to deliver good performance under partially replicated scenarios. Once again, we can verify the same patterns in scenarios 7 and 8 as in scenarios 4 and 5, respectively. The lower throughput values in these scenarios, is due to the fact that, as the use case application is partially replicated, microservices need to contact other instances that are further away, resulting in higher communication latency between microservices and in longer request processing times. This phenomenon can be considered as an artifact of the experience setup.

Now we will analyze, what happens in terms of latency. Figure 4.8 presents the latency (the y-axis, in milliseconds) obtained in the scenarios 1 and 2 over the period of five minutes of the experience (in the x-axis) in the first test for both Sheik and Eureka

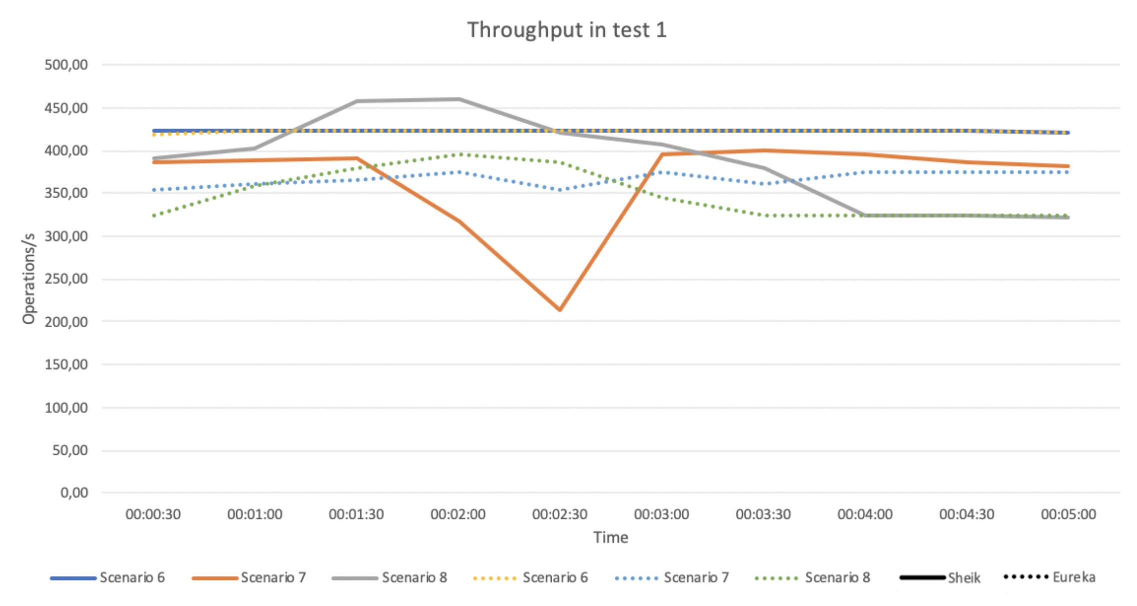

Figure 4.7: Throughput over time in scenarios 6, 7, and 8 in test 1.

(differentiated by a continuous and dotted line, respectively). The results show a low and steady latency throughout the execution of the test for both systems.

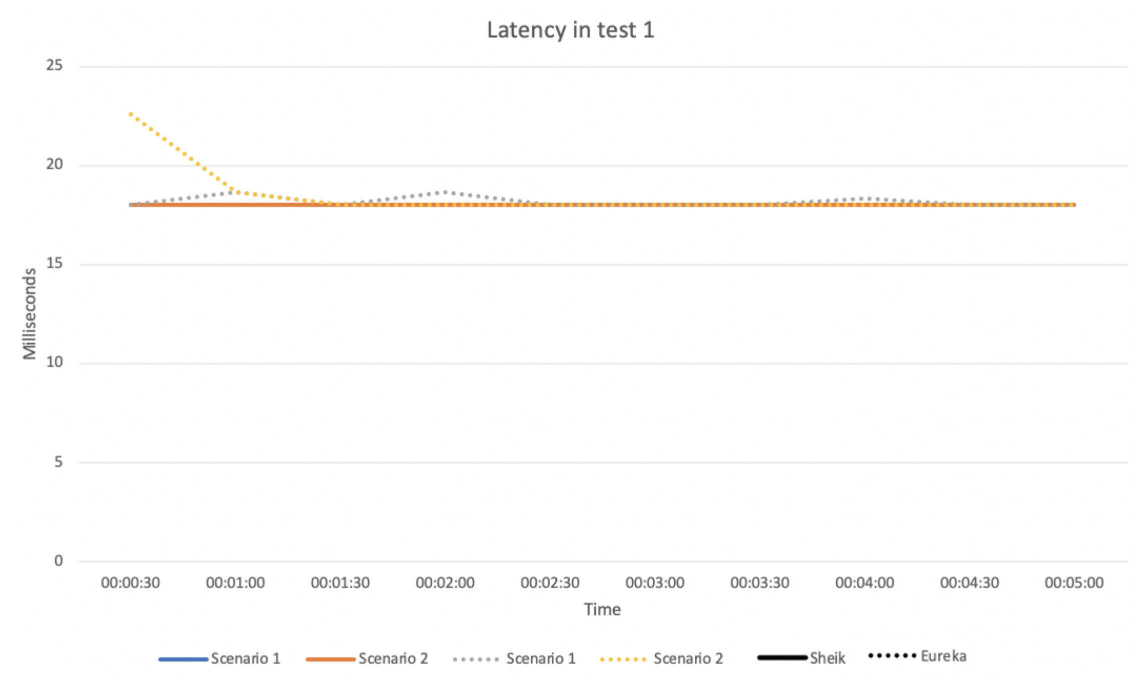

Figure 4.8: Latency over time in scenarios 1 and 2 in test 1.

Figure 4.9 presents the latency (in the y-axis, in milliseconds) obtained in the scenarios 3, 4 and 5 over the five minutes of the experience (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results reinforce the results obtained for the throughput, where in scenario 4 we have a spike in latency occurring at minute 02:30, resulting from the microservices executing the failover protocol, and in scenario 5 the latency first gets lower, as microservices are unable to communicate with the failed instances which results in (fast) requests failing, and as the system stabilizes, the latency increases due to microservices now communicating with instances that are further away, and thus having higher communication latency due to that distance.

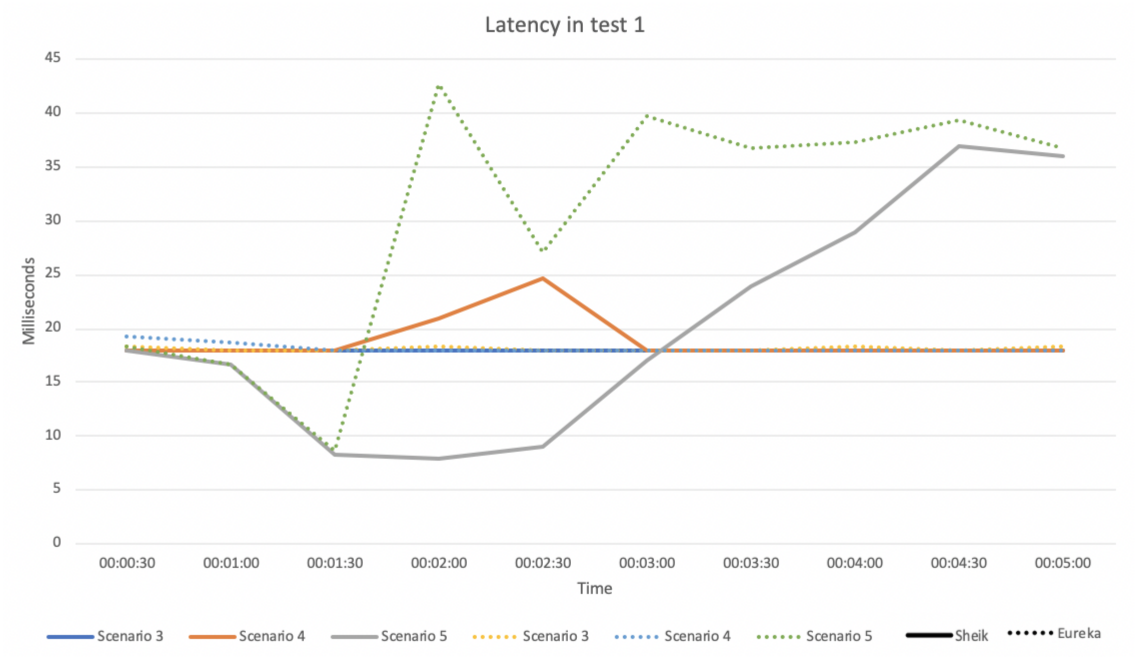

Figure 4.9: Latency over time in scenarios 3, 4, and 5 in test 1.

Finally, Figure 4.10 presents the latency (in the y-axis, in milliseconds) obtained in the scenarios 6, 7 and 8 over the time of the experience (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). In these scenarios, as the use case application is partially replicated across the three clusters (two of which are on opposite sides of France), the latency is considerably higher when compared with the previous scenarios. This result is expected as microservices need to make several requests to other microservices (the orders microservice needs to contact the payment and shipping microservices, that in turn need to contact the user microservice) some of which might be located might be located in remote locations, with higher latency, which delays the processing of clients requests. In these scenarios, latency displays the same behaviour as the corresponding throughput results, displaying a steady latency of 47 ms in scenario 6 for both solutions. In scenario 7, Sheik has a spike in latency at minute 02:30 of the experience and quickly stabilizes at 50 ms, while Eureka displays a slightly variable latency that has values between 53 ms and 55 ms. In scenario 8, both Sheik and Eureka have a similar behaviour, as when the first microservice fails at minute 01:00 the latency drops, which results from other microservices being unable to complete requests and failing, and, as the system stabilizes, the latency starts to rise as requests start to be redirected to other microservice instances that are further away from the requester.

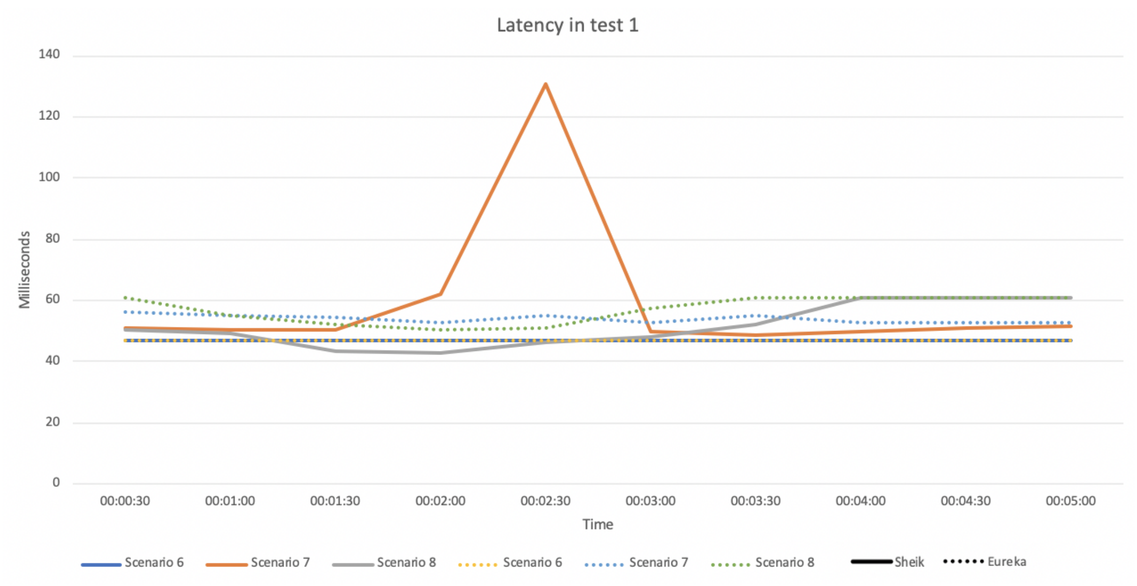

Figure 4.10: Latency over time in scenarios 6, 7, and 7 in test 1.

We present the results for the experiences with a longer duration in Annex I.

Next, we present the performance of both solutions obtained in the third test for each scenario. Tho objective of this tests is to understand how the solutions behave under a heavier load, thus the number of clients making requests is doubled. Figure 4.11 presents the throughput(in the y-axis, in operations per second) obtained in the first five minutes (in the x-axis) in the first test for both Sheik and Eureka (differentiated by a continuous and dotted line,respectively). The result show that, in both scenarios, Sheik has a better and more stable performance than Eureka, with Eureka displaying an even more accentuated variation in values.

Figure 4.12 presents the throughput (in the y-axis, in operations per second) obtained in the scenarios 3, 4 and 5 during the five minutes of the experience (in the x-axis) in the third test for both Sheik and Eureka (differentiated by a continuous and dotted line,respectively). The results show that both Sheik and Eureka demonstrate an equivalent performance in scenario 3. In scenario 4, Sheik displays a slightly lower performance than Eureka, due to the overhead caused by the failover protocol (as Sheik employs a partial replication strategy contrary to Eureka's full replication), which confirms the results obtained above. In scenario 5 however, Sheik demonstrates a higher throughput. Although in both cases there are requests that fail, Sheik achieves better latency in those requests that do not fail.

Finally, Figure 4.13 presents the throughput (in the y-axis, in operations per second) obtained in the scenarios 6, 7 and 8 over the time of the experience (in the x-axis) in the third test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). In scenario 6, both solution display an equivalent performance with a lower throughput than in the previous tests, as the delay introduced by the datacenters

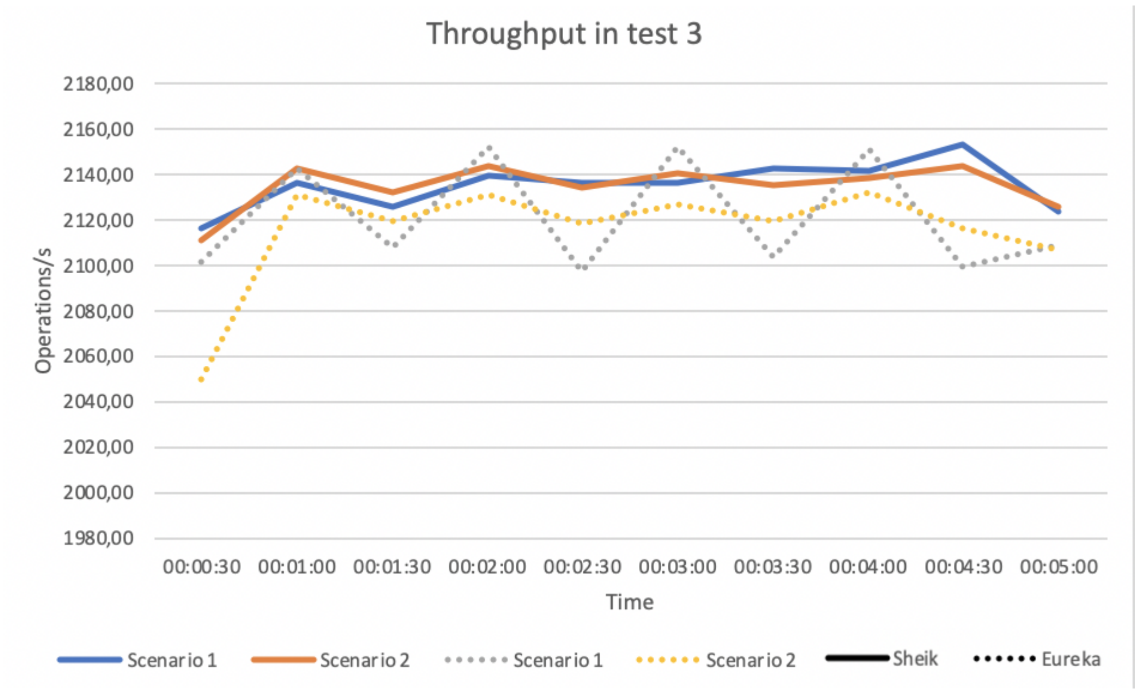

Figure 4.11: Throughput over time in scenarios 1 and 2 in test 3.

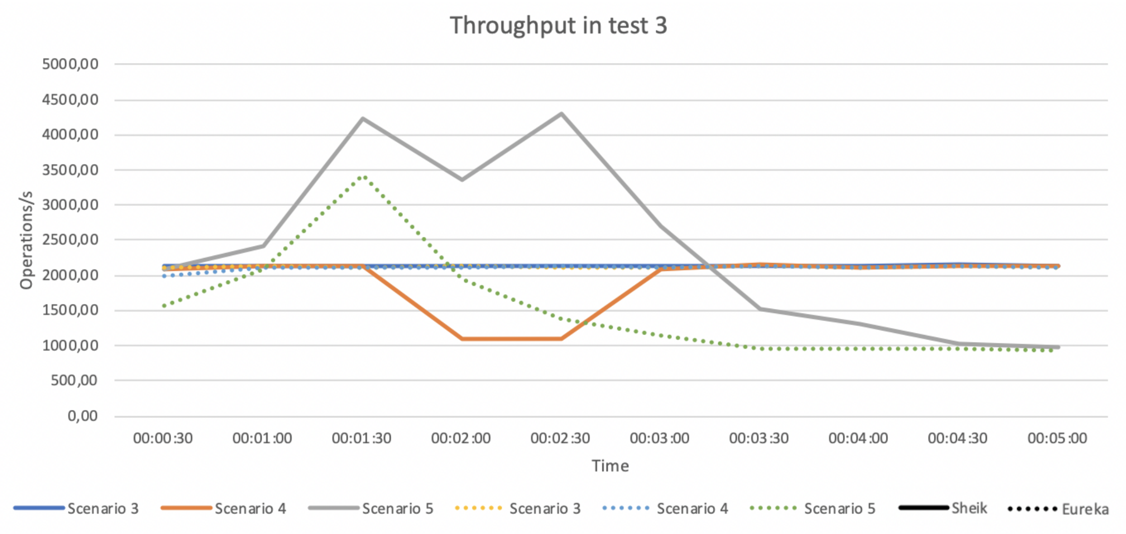

Figure 4.12: Throughput over time scenarios 3, 4, and 5 in test 3.

is predominant, regarding the latency of contacting Sheik or Eureka, adding to the delays introduced by processing the requests on the services side. In scenario 7, Sheik displays a slightly lower performance than Eureka, due to the overhead caused by the failover protocol when a Sheik instance fails. In scenario 8, Sheik demonstrates a slightly higher throughput, because it shows a lower latency than Eureka when the impact of the microservice instances failing seems to be equal.

Now we will analyze, what happens in terms of latency. Figure 4.14 presents the latency (the y-axis, in milliseconds) obtained in the scenarios 1 and 2 over the period of

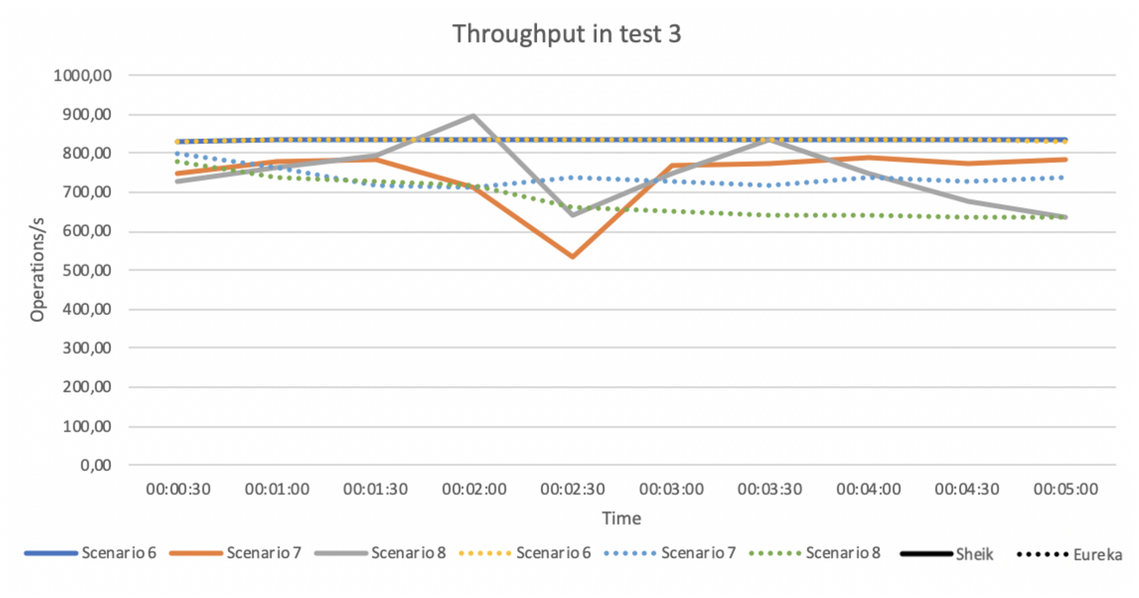

Figure 4.13: Throughput over time scenarios 6, 7, and 8 in test 3.

five minutes of the experience (in the x-axis) in the third test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show a low and steady latency throughout the execution of the test for both systems, although Eureka has some spikes.

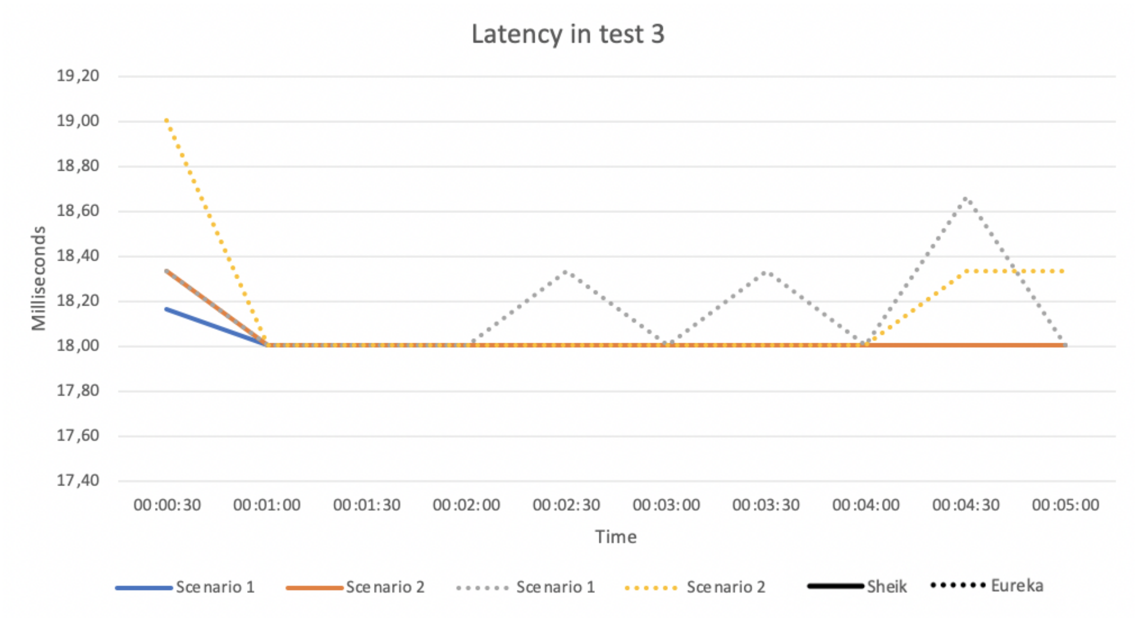

Figure 4.14: Latency over time in scenarios 1 and 2 in test 3.

Figure 4.15 presents the latency (in the y-axis, in milliseconds) obtained in the scenarios3, 4 and 5 over the five minutes of the experience (in the x-axis) in the third test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results confirm the results obtained for the corresponding throughput, where in scenario

3 both solutions are equivalent, with Sheik having a slightly higher latency in scenario 4 (due to the overhead caused by the failover protocol) and a lower latency in scenario 5.

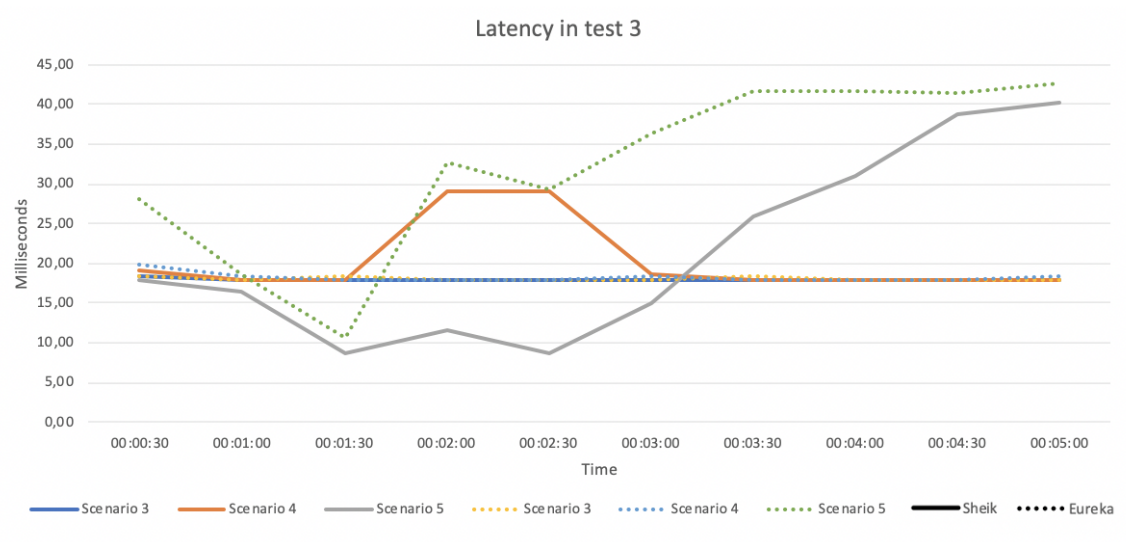

Figure 4.15: Latency over time in scenarios 3, 4, and 5 in test 3.

Finally, Figure 4.16 presents the latency (in the y-axis, in milliseconds) obtained in the scenarios 6, 7 and 8 over the time of the experience (in the x-axis) in the third test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). In these scenarios, as the use case application is partially replicated, the latency is considerably higher when compared with the previous scenarios and displays the same behaviour as the corresponding throughput results. In scenario 6 both solution display a steady latency of 50ms, with Sheik peaking at 85ms in scenario 7 contrary to the 65ms of Eureka. In scenario 8, both solutions display the same behaviour, with Sheik's latency being slightly lower overall.

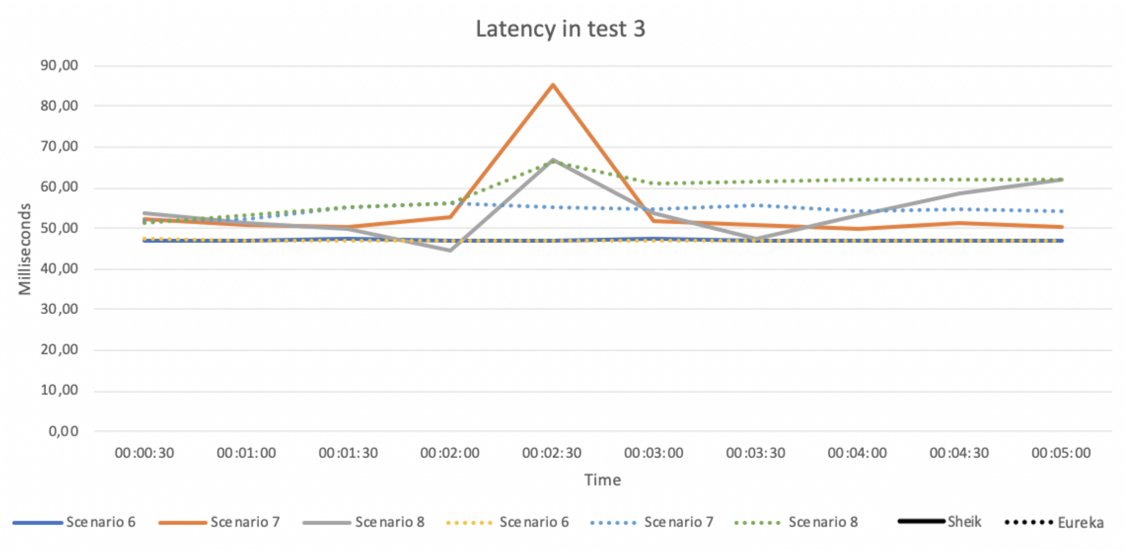

Figure 4.16: Latency over time in scenarios 6, 7, and 8 in test 3.

4.4.3 Scalability

In this Section, we are going to present Sheik's scalability results when compared to those of Eureka. In order to obtain these results, we conducted several tests in two scenarios (scenario 3, that is fully replicated, and scenario 6, that is partially replicated), with an increasing number of clients making concurrent requests (1, 2, 4, 6, 8, 10, and 12 JMeter threads) to generate the load and extract the performance metrics.

Figure 4.17 shows the results obtained for the throughput (in the left y-axis, in operations per second) and the latency (in the right y-axis, in milliseconds) for the scenario 3 during five minutes of execution for a several number of concurrent clients (in the x-axis) for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show that both solutions have an equivalent performance both in terms of throughput and latency.

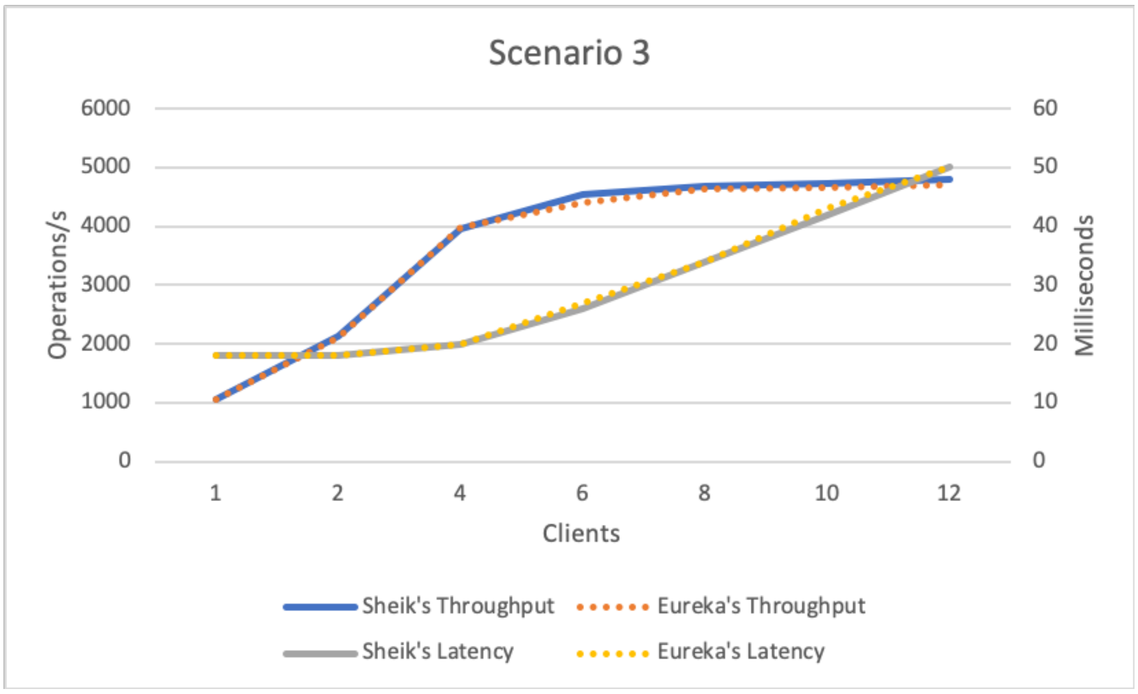

Figure 4.17: Scalability performance in scenario 3.

Figure 4.17 shows the results obtained for the throughput (in the left y-axis, in operations per second) and the latency (in the right y-axis, in milliseconds) for the scenario 6 during five minutes of execution for a several number of concurrent clients (in the x-axis) for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show that both solutions have a mostly equivalent performance, although it's noticeable that Sheik's latency is slightly lower and the throughput is slightly higher than Eureka's.

It is important to note that, this experiment needed to be performed with more microservices and locations, which was not possible due to the unavailability of enough nodes, as it is possible that both solutions are not yet fully saturated and it is very likely

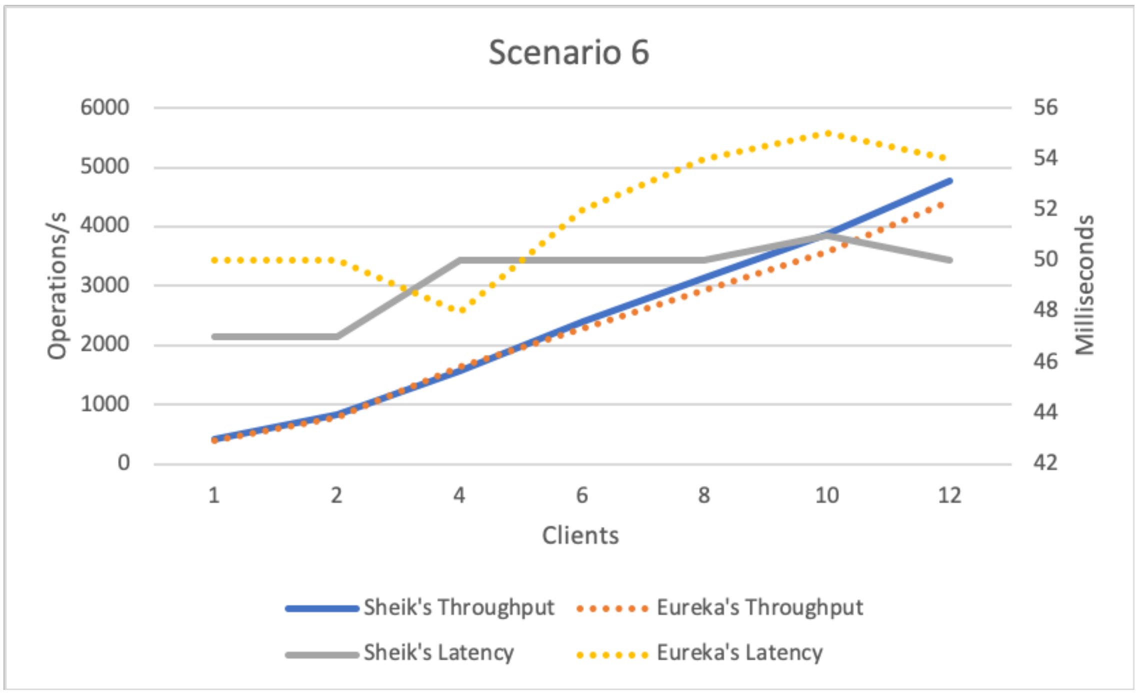

Figure 4.18: Scalability performance in scenario 6.

that Eureka´s full replication strategy will damage its performance.

4.5 Summary

In this Chapter we have presented our experimental validation and evaluation of Sheik, where we presented an overall performance and overhead evaluation of Sheik when compared to Eureka, a more detailed performance evaluation of our solution by analyzing its performance over time, when compared against the baseline solution, and a scalability study. Overall, Sheik demonstrated to be a robust solution that is able to perform well under several deployment conditions, being able to adequately deal with both microservice instances and other Sheik instances fails.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The edge computing paradigm emerged as a strategy to complement cloud architectures and address the existing limitations of cloud infrastructures. As the number of devices increase, and consequently, the data being produced and in need of processing grows larger, cloud infrastructures are rapidly becoming unable to process and produce responses in a timely fashion. The resulting hybrid cloud/edge infrastructures however, are highly heterogeneous due to a high number of very diverse edge resources, some of which with limited computational and storage power.

A way to simplify the development of robust cloud/edge applications is through Microservice Architecture (MSA), that allows to decompose applications into a set of small, independent, and single purpose services that can be developed independently. Microservice applications however, tend to grow into applications with thousands and hundreds of services, which are very hard to manage since microservices need to communicate with each other, and in order to be able to interact they need to know each others network locations, a task that is not easily solved when a static configuration is not viable and microservices are frequently being launched and deactivated across the cloud/edge spectrum.

In this work, we focused on the challenge of developing a microservice discovery service, that simplifies the interactions between microservices by providing a registration and discovery service that allows to dynamically bind and locate microservices in the cloud/edge settings.

5.2 Future Work

As future work we want to conduct an experimental evaluation of Sheik in larger scale settings, containing a larger number of instances, in a real hybrid cloud/edge environment. Additionally, the work developed in this thesis, allowed to make an important and fundamental step into further developing Triforce, a middleware solution that dynamically and autonomically will handle the complexity of monitoring, migrating, and replicating miscroservices in the hybrid cloud/edge ecosystem.

Bibliography

[1] *Akamai*. https://www.akamai.com/. Accessed: 14-02-2019.

[2] *Akamai Cloudlets*. https://www.akamai.com/us/en/products/performance/cloudlets/. Accessed: 11-02-2019.

[3] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98.

[4] *Amazon EC2*. https://aws.amazon.com/ec2/features/. Accessed: 30-01-2019.

[5] *Apache JMeter*. https://jmeter.apache.org/. Accessed: 19-02-2019.

[6] *Apache Kafka*. https://kafka.apache.org/. Accessed: 18-02-2019.

[7] *Apache Thrift*. https://thrift.apache.org/. Accessed: 19-04-2020.

[8] *Apache Tomcat*. http://tomcat.apache.org/. Accessed: 11-03-2020.

[9] *Apache ZooKeeper*. https://zookeeper.apache.org/. Accessed: 18-02-2019.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A view of cloud computing." In: *Commun. ACM* 53.4 (2010), pp. 50–58. doi: 10.1145/1721654.1721672. url: http://doi.acm.org/10.1145/1721654.1721672.

[11] *AWS Elastic Load Balancing*. https://aws.amazon.com/elasticloadbalancing/. Accessed: 19-04-2020.

[12] *AWS Lambda*. https://aws.amazon.com/lambda/features/. Accessed: 11-02-2019.

[13] *AWS Lambda@Edge*. https://aws.amazon.com/lambda/edge/. Accessed: 20-02-2019.

[14] *Azure Service Bus*. https://azure.microsoft.com/en-gb/services/service-bus/. Accessed: 19-04-2020.

[15] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. "Adding Virtualization Capabilities to the Grid'5000 Testbed." In: *Cloud Computing and Services Science*. Ed. by I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan. 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.

[16] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan. "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers." In: *Computer Networks* 130 (2018), pp. 94–120. DOI: 10.1016/j.comnet.2017.10.002. URL: https://doi.org/10.1016/j.comnet.2017.10.002.

[17] F. Bonomi, R. A. Milito, J. Zhu, and S. Addepalli. "Fog computing and its role in the internet of things." In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*. 2012, pp. 13–16. DOI: 10.1145/2342509.2342513. URL: https://doi.org/10.1145/2342509.2342513.

[18] A. Botta, W. De Donato, V. Persico, and A. Pescapé. "Integration of cloud computing and internet of things: a survey." In: *Future generation computer systems* 56 (2016), pp. 684–700.

[19] E. A. Brewer. "Towards robust distributed systems (abstract)." In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. 2000, p. 7. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502.

[20] R. Buyya and A. V. Dastjerdi. "Internet of Things: Principles and paradigms." In: Elsevier, 2016. Chap. 16.3.3.

[21] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting. "A Fragment-based Approach for Efficiently Creating Dynamic Web Content." In: *ACM Trans. Internet Technol.* 5.2 (May 2005), pp. 359–389. ISSN: 1533-5399. DOI: 10.1145/1064340.1064343. URL: http://doi.acm.org/10.1145/1064340.1064343.

[22] *Cisco Global Cloud Index: Forecast and Methodology, 2014–2019*. Tech. rep. CISCO, 2015.

[23] *Consul*. https://www.consul.io/. Accessed: 20-04-2020.

[24] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform." In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's globally distributed database." In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[26] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)* International computer science series. Addison-Wesley-Longman, 2002. ISBN: 978-0-201-61918-8.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[28] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. "Epidemic algorithms for replicated database maintenance." In: *ACM SIGOPS Operating Systems Review* 22.1 (1988), pp. 8–32.

[29] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: yesterday, today, and tomorrow." In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[30] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan. "The Case for Mobile Edge-Clouds." In: *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing, UIC/ATC 2013, Vietri sul Mare, Sorrento Peninsula, Italy, December 18-21, 2013*. 2013, pp. 209–215. DOI: 10.1109/UIC-ATC.2013.94. URL: https://doi.org/10.1109/UIC-ATC.2013.94.

[31] M. Eltoweissy, S. Olariu, and M. F. Younis. "Towards Autonomous Vehicular Clouds - A Position Paper (Invited Paper)." In: *Ad Hoc Networks - Second International Conference, ADHOCNETS 2010, Victoria, BC, Canada, August 18-20, 2010, Revised Selected Papers*. 2010, pp. 1–16. DOI: 10.1007/978-3-642-17994-5_1. URL: https://doi.org/10.1007/978-3-642-17994-5_1.

[32] *eShopOnContainers Application*. https://github.com/dotnet-architecture/eShopOnContainers. Accessed: 19-02-2019.

[33] *etcd*. https://github.com/etcd-io/etcd. Accessed: 19-04-2020.

[34] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The many faces of publish/subscribe." In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[35] *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. Tech. rep. CISCO, 2015.

[36] *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*. https://www.infoq.com/presentations/linkedin-microservices-urn/. Accessed: 18-04-2020.

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

[38] *Google App Engine*. https://cloud.google.com/appengine/. Accessed: 30-01-2019.

[39] M. Kalske, N. Mäkitalo, and T. Mikkonen. "Challenges when moving from monolith to microservice architecture." In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47.

[40] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. *Towards Enabling Novel Edge-Enabled Applications*. Tech. rep. DI-FCT-UNL, 2018.

[41] J. Leitão, M. C. Gomes, N. Preguiça, P. Á. Costa, V. Duarte, A. Carrusca, and A. Lameirinhas. *A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications*. Tech. rep. DI-FCT-UNL, 2018.

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

[43] J. Leitao, J. Pereira, and L. Rodrigues. "Gossip-based broadcast." In: *Handbook of Peer-to-Peer Networking*. Springer, 2010, pp. 831–860.

[44] J. Leitão. "Topology Management for Unstructured Overlay Networks." Master's thesis. Technical University of Lisbon.

[45] M. Letia, N. Preguiça, and M. Shapiro. "CRDTs: Consistency without concurrency control." In: *arXiv preprint arXiv:0907.0929* (2009).

[46] *leveldb*. https://github.com/google/leveldb. Accessed: 20-04-2020.

[47] J. Lewis and M. Fowler. *Microservices, a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html.

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

[49] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions." In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 283–292.

[50] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 401–416.

[51] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage." In: *NSDI*. Vol. 13. 2013, pp. 313–328.

[52] T. H. Luan, L. X. Cai, J. Chen, X. Shen, and F. Bai. "VTube: Towards the media rich city life with autonomous vehicular content distribution." In: *Proceedings of the 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2011, June 27-30, 2011, Salt Lake City, UT, USA*. 2011, pp. 359–367. DOI: 10.1109/SAHCN.2011.5984918. URL: https://doi.org/10.1109/SAHCN.2011.5984918.

[53] P. Mell, T. Grance, et al. *SP 800-145. The NIST definition of cloud computing*. Tech. rep. National Institute of Standards and Technology (NIST), 2011.

[54] *Microsoft Windows Azure Cloud Services*. https://azure.microsoft.com/en-gb/services/cloud-services/. Accessed: 30-01-2019.

[55] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. "Architectural styles, design patterns, and objects." In: *IEEE software* 14.1 (1997), pp. 43–52.

[56] *Neflix Platform Engineering — we're just getting started*. https://netflixtechblog.com/neflix-platform-engineering-were-just-getting-started-267f65c4d1a7. Accessed: 16-04-2020.

[57] *Netflix Eureka*. https://github.com/Netflix/eureka. Accessed: 18-02-2019.

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

[59] G. Orsini, D. Bade, and W. Lamersdorf. "Computing at the Mobile Edge: Designing Elastic Android Applications for Computation Offloading." In: *8th IFIP Wireless and Mobile Networking Conference, WMNC 2015, Munich, Germany, October 5-7, 2015*. 2015, pp. 112–119. DOI: 10.1109/WMNC.2015.10. URL: https://doi.org/10.1109/WMNC.2015.10.

[60] Z. Pang, L. Sun, Z. Wang, E. Tian, and S. Yang. "A Survey of Cloudlet Based Mobile Computing." In: *International Conference on Cloud Computing and Big Data, CCBD 2015, Shanghai, China, November 4-6, 2015*. 2015, pp. 268–275. DOI: 10.1109/CCBD.2015.54. URL: https://doi.org/10.1109/CCBD.2015.54.

[61] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. "Service-oriented computing: State of the art and research challenges." In: *Computer* 40.11 (2007), pp. 38–45.

[62] *RabbitMQ*. https://www.rabbitmq.com/. Accessed: 18-02-2019.

[63] C. Richardson and F. Smith. *Microservices: From Design to Deployment*. NGINX, 2016.

[64] A. Rodriguez. "Restful web services: The basics." In: *IBM developerWorks* 33 (2008), p. 18.

[65] M. Satyanarayanan. “The emergence of edge computing.” In: *Computer* 50.1 (2017), pp. 30–39.

[66] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. “The Case for VM-Based Cloudlets in Mobile Computing.” In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1536-1268. DOI: 10.1109/MPRV.2009.82. URL: http://dx.doi.org/10.1109/MPRV.2009.82.

[67] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. “Serendipity: Enabling Remote Computing Among Intermittently Connected Mobile Devices.” In: *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*. MobiHoc ’12. Hilton Head, South Carolina, USA: ACM, 2012, pp. 145–154. ISBN: 978-1-4503-1281-3. DOI: 10.1145/2248371.2248394. URL: http://doi.acm.org/10.1145/2248371.2248394.

[68] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. “Edge Computing: Vision and Challenges.” In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://doi.org/10.1109/JIOT.2016.2579198.

[69] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. “Time-aware reactive storage in wireless edge environments.” In: *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. 2019, pp. 238–247.

[70] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. “Analysis of Caching and Replication Strategies for Web Applications.” In: *IEEE Internet Computing* 11.1 (2007), pp. 60–66. DOI: 10.1109/MIC.2007.3. URL: https://doi.org/10.1109/MIC.2007.3.

[71] D. Taibi, V. Lenarduzzi, and C. Pahl. “Architectural patterns for microservices: a systematic mapping study.” In: SCITEPRESS, 2018.

[72] C. Tang, R. N. Chang, and C. Ward. “GoCast: Gossip-enhanced overlay multicast for fast and dependable group communication.” In: *2005 International Conference on Dependable Systems and Networks (DSN’05)*. IEEE. 2005, pp. 140–149.

[73] *the morning paper: An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems*. https://blog.acolyer.org/2019/05/13/an-open-source-benchmark-suite-for-microservices-and-their-hardware-software-implications-for-cloud-edge-systems/. Accessed: 18-05-2020.

[74] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.” In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

[75] A. Wang and S. Tonse. *Announcing Ribbon: Tying the Netflix Mid-Tier Services Together*. 2013. URL: https://medium.com/netflix-techblog/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62.

[76] S. Yi, Z. Hao, Z. Qin, and Q. Li. "Fog Computing: Platform and Applications." In: *Third IEEE Workshop on Hot Topics in Web Systems and Technologies, HotWeb 2015, Washington, DC, USA, November 12-13, 2015*. 2015, pp. 73–78. DOI: 10.1109/HotWeb.2015.22. URL: https://doi.org/10.1109/HotWeb.2015.22.

[77] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: state-of-the-art and research challenges." In: *J. Internet Services and Applications* 1.1 (2010), pp. 7–18. DOI: 10.1007/s13174-010-0007-6. URL: https://doi.org/10.1007/s13174-010-0007-6.

[78] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. "Peer-assisted content distribution in akamai netsession." In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 31–42.

Annex I

Annex 1 - Extra performance evaluation

In this Annex, we present the performance of both solutions obtained in the second test, for each of the eight scenarios. The objective of the second test is to better understand how the solutions will behave under a longer period of execution time, thus this test has a duration of 10 minutes. Similarly to the tests discussed previously, the results will be divided into three groups for better readability. Figures I.1, I.2, and I.3 present the throughput (in the y-axis, in operations per second) obtained in all the scenarios during the 10 minutes of the experience (in the x-axis) in the second test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results show that both solutions are able to operate steadily for long periods of time. All scenarios display the same behaviour that was described in test 1, and after normalizing, both solutions show a prescribed behaviour with no further disturbances, which shows that the systems normalize correctly.

Figures I.4, I.5, and I.6 present the latency (in the y-axis, in milliseconds) obtained in all the scenarios during the 10 minutes of the experience (in the x-axis) in the second test for both Sheik and Eureka (differentiated by a continuous and dotted line, respectively). The results obtained confirm the results obtained for the throughput, and show a generally stable latency through the execution of the test. Similarly to the throughput, each scenario displays the same behaviour as in the first test, and after the system normalizes from the instance's fails, both solutions display a stable latency, although in scenario 2, Eureka's spikes in latency in the beginning of the test are more noticeable than in test 1.

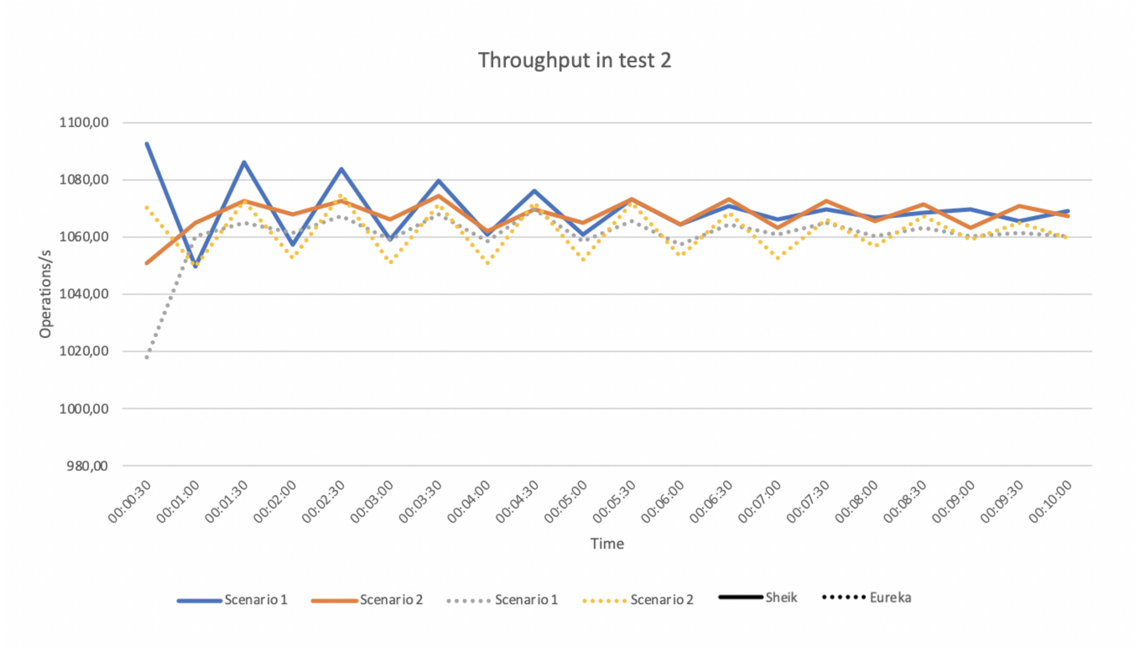

Figure I.1: Throughput over time in scenarios 1 and 2 in test 2.

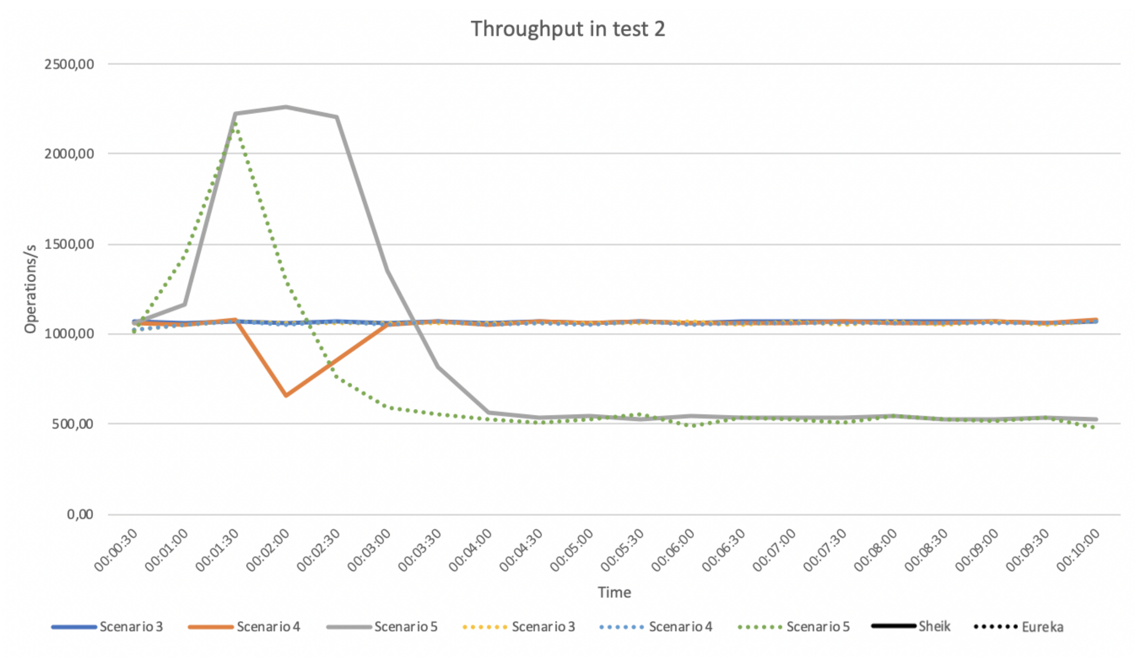

Figure I.2: Throughput over time in scenarios 3, 4, and 5 in test 2.

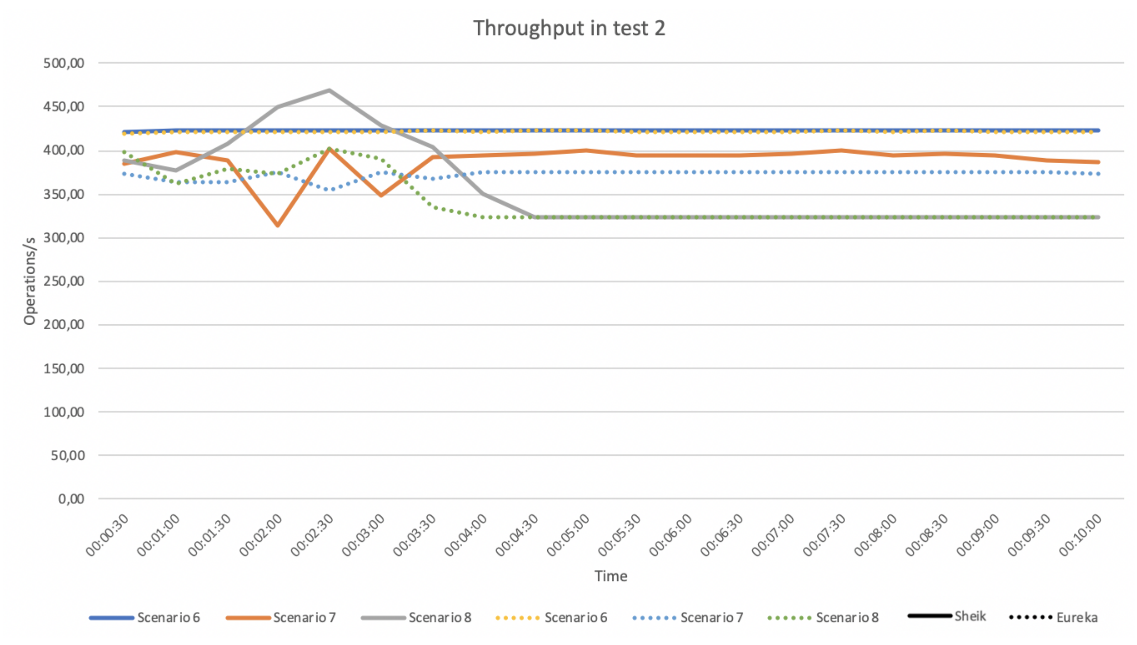

Figure I.3: Throughput over time in scenarios 6, 7, and 8 in test 2.

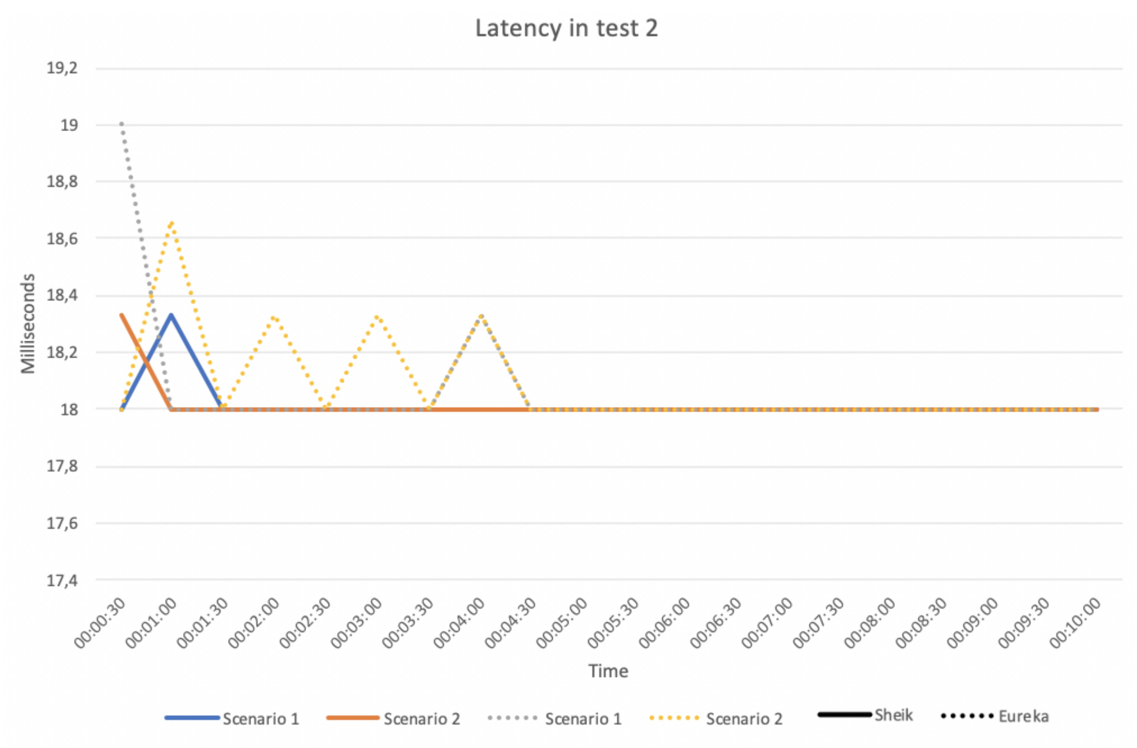

Figure I.4: Latency over time in scenarios 1 and 2 in test 2.

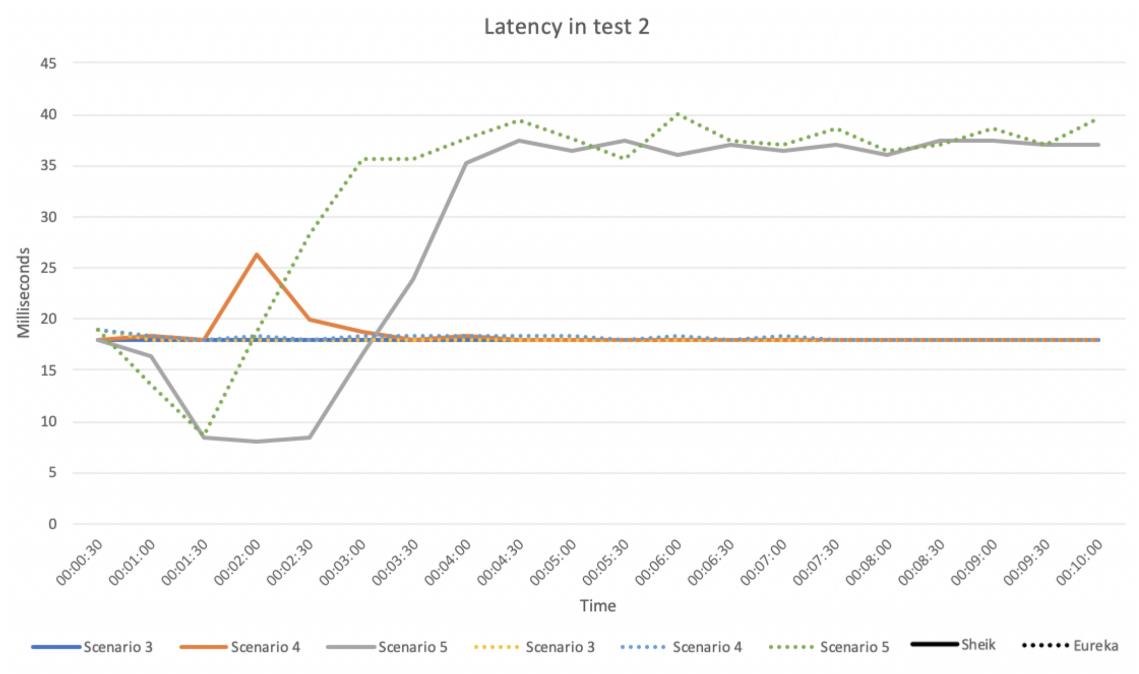

Figure I.5: Latency over time in scenarios 3, 4, and 5 in test 2.

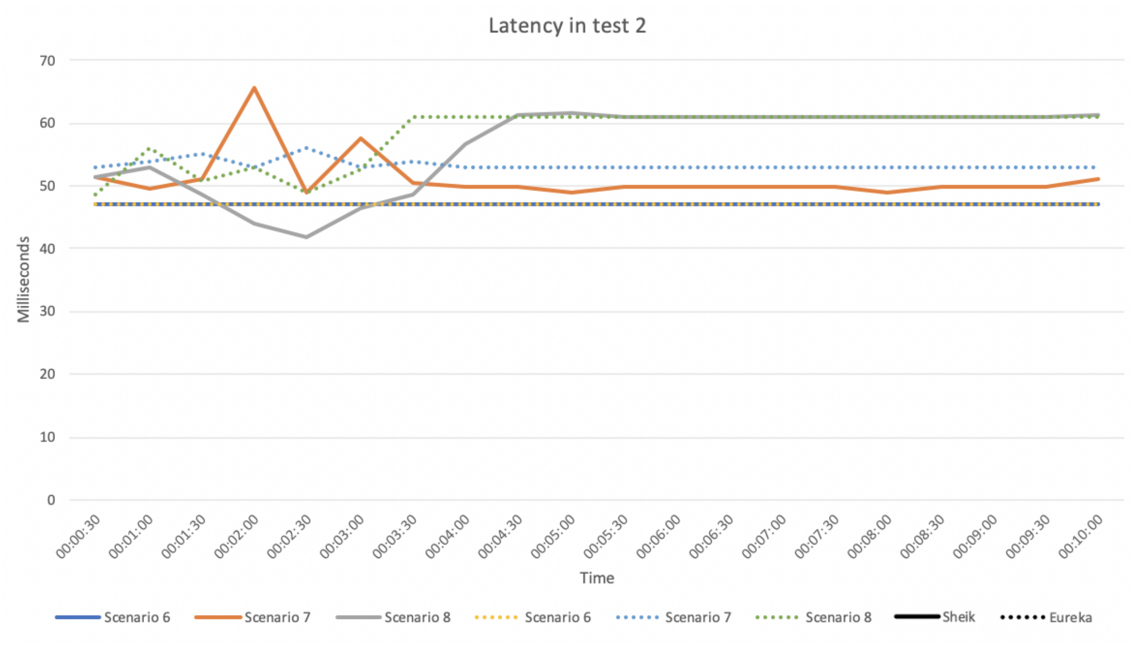

Figure I.6: Latency over time in scenarios 6, 7, and 8 in test 2.