Khrystyna Fedyuk

Degree in Computer Science and Engineering

Triforce: A Middleware Support to enable Dynamic Micro-Services in the Edge

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

Master of Science in
Computer Science and Engineering

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

Co-adviser: Maria Cecília Gomes, Assistant
Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2019

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 pats 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 have limited computational and storage power. As a result, cloud/edge systems must be very dynamic and agile in order to adapt to the constantly evolving execution conditions.

Thus, in this document, we aim at developing a middleware solution, called Triforce, that simplifies the construction of efficient and robust cloud/edge applications 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. Moreover, the middleware will dynamically and autonomically handle the complexity of monitoring, migrating, and replicating miscroservices in the hybrid cloud/edge environment.

Keywords: Edge Computing, Microservices, Middleware

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 futuro 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 no limite 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/limite resultante, no entanto, é altamente heterogênea devido a um grande número de recursos altamente diversos presentes no limite, alguns dos quais têm capacidade computacional e de armazenamento limitada. Como resultado, esses sistemas híbridos devem ser muito ágeis e dinâmicos para se poderem adaptar às condições de execução em constante evolução.

Assim, neste documento, pretendemos desenvolver uma solução de middleware, chamada Triforce, que simplifica a construção de aplicações híbridas eficientes e robustas tomando partido de arquiteturas de microsserviços, que permitem decompor as aplicações num conjunto de pequenos e independentes serviços, com finalidade única, que podem ser desenvolvidos de forma independente. Além disso, o middleware vai lidar com a complexidade de monitorização, migração e replicação de microsserviços no ambiente de nuvem/limite híbrido de uma forma dinâmica e autônoma.

Palavras-chave: Computação no Limite, Microserviços, Middleware

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 [72]. Highly scalable and elastic, cloud computing infrastructures allow dynamic allocation of seemingly unlimited resources, adapting to the 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 [38]. 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 [61] 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, 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 [38].

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 [37]. A way to simplify the development and integration of such applications is through Microservice Architectures (MSA) [24], where each microservice is a small, independent, and single purpose service, whose combination allow to define the applications.

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.

Problem Definition

When talking on a small scale, like a smart home, MSA is the perfect solution for building efficient and robust applications for the cloud/edge platforms. But as we consider larger-scale applications, as in a smart city setting for example, the large number microservices deployed across an evenly large number of heterogeneous devices becomes too hard to manage manually, since decisions like where and what to replicate becomes too complex (as they depend on multiple factors). If we try to expand the setting to the whole globe (e.g. user-centric applications), microservices become geo-replicated. 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 manual management of such applications becomes unfeasible in practice.

To tackle this problem, we need to be able to dynamically control key aspects, that are not totally intdependent: *i)* the application logic plane, that translates into controlling the life-cycle of individual instances of miscroservices; and *ii)* the application data plane, that translates into controlling the life-cycle of data storage replicas used by the microservices. Adittionally, an adaptive monitoring system that can effectively gather relevant information and provide this information to all components in the system that make re-configuration decisions will also be required.

Taking this into account, the need of an integrated solution emerged. Thus, in this thesis, we propose a middleware for creating autonomic Microservice Architectures for cloud/edge environments, called Triforce, that deals with most of the complexity of the management by monitoring, moving, and replicating microservices dynamically, with little human interference.

Structure of the Document

The rest of the document is organized as follows:

- In chapter 2 we provide a more complete explanation of microservices, virtualization and other relevant building blocks, and the three key aspects of our middleware, presenting the current existing solutions for each of them and how can we leverage on them for our solution.

- Chapter 3 briefly presents our proposed solution, Triforce, and details the approach for achieving it. It also includes a planning session that details the schedualing of 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 (2.1) and the technologies that exist in the edge (2.2) to be able to make the parallel between the two. Additionally, replication (2.3) and monitoring (2.4) strategies will be discussed in order to able to extend the applications from the cloud to the edge environment. Finally, some helpful building blocks (2.5) will be discussed as a way to simplify and facilitate the deployment of these applications to the various execution environments present in the cloud/edge infrastructure.

2.1 Cloud Computing

Cloud computing appeared as a solution for the long lasting dream of computing as a utility [8]. 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 molding 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 [72].

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) [8, 64].

Using an external definition, and in summary, according to The National Institute of Standards and Technology (NIST) [48], 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 [48, 72]:

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 [48, 72].

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 [13, 72]:

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) [48, 72]:

- 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) agreed with the customers, that it is mandatory to fulfill.
- 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 what gives 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 [8, 72]:

- 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 the peak load.

- 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 [32] 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 [49], 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 [17], 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 the IoT and mobile applications led to the emergence of edge computing, a new paradigm where data processing occurs, in part, at the edge of the network instead of occurring completely 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 [9]. 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 [10], and thus reduce latency and improve performance.

Cloudlet

The cloudlet concept was introduced in 2009 by Satyanarayanan et al. [62] as a solution to address mobile device's resource limitations, namely their lower amounts of memory and minited 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 [52], 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 [30], fog computing tries to bring computation closer to the things that produce and act on IoT data by putting computing capabilities in the connection 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 [11].

Fog computing has a broad range of applications such as in: *i)* wireless sensor and actuator networks [12], 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 [71], 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 [46] 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 [27].

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 [11]. 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. [25], 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 [63] 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 [51] 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.

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. 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 a problem 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 and be 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 how to replicate the data and the logical part of the application, and perform effective monitoring of workloads and access patterns over applications.

2.3 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 [20].

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. 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.

2.3.1 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 fourth 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.

2.3.2 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 will later explain in the Section 2.5, 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. Therefore it is necessary to deal with the replication of these computational components, employing active multi-master replication, which entails controlling the life-cycle of individual components, including where new instances should be deployed and how they interconnect among each other.

Another concern is that, 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, and possibly modify the binding between them to ensure efficient communication or, if sets of instances are communicating frequently, it may be beneficial to replicate them together to locations in close vicinity, as to promote low latency access.

2.3.2.1 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 [65]. 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 [15]. These features can also be implemented by leveraging lambda functions, that can inject content to web pages depending on the location of the user.

2.3.3 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 data replicas 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 [14]. 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 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 contradictory 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.

2.3.3.1 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 [21] 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) [44] 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 [45] 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 [18] 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 [19] 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.

2.3.3.2 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 [73] 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 [43], that allows to replicate portions of the state of a web application (data objects) in the form of CRDTs [40] 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.

2.3.4 Discussion

Replication is an essential mechanism that will allow to elastically expand the application data layer and the application logic layer 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 the application and the data layers. 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. It's important to note that, application logic component are replicated in response to the service demand, and application data components are replicated according to the application logic components needs.

Additionally, in order to take full advantage of the edge infrastructure, components need to be executed in the most appropriate edge resource. Deciding the ideal location to execute a particular component can be very tricky, since it depends on multiple factors, some of which are dynamic and might change frequently (for instance the component access pattern, its resource consumption, and the availability of computational resources at different locations). To better capture the state of the system and enable better replication management decisions, next, we will discuss monitoring solutions.

2.4 Monitoring

Monitoring is essential to operate any large-scale distributed system as it is an essential aspect to ensure that the service provides favourable QoS, such as low request processing latency and high availability. In order to guarantee these properties it is necessary to monitor the infrastructure in terms of resource consumption (CPU, Memory, Bandwidth, Number of Disk Operations) in order to detect when the device is overloaded, and the application in terms of performance (number of processed operations, average response time) and metrics related with the operation of the application (number of received requests and their origin) in order to detect degradation of quality of service.

Based on these metrics, a monitoring system can then take decisions regarding when, where and what to replicate: *i)* if the infrastructure is overloaded its a sign that the component needs to be replicated; *ii)* if the service is receiving a lot of requests then one can reason its the most useful service to replicate; *iii)* knowing from where the requests are arriving it's a clue as to where to replicate; and *iv)* quality metrics such as latency and availability are an extra indicator that services need to be replicated (or migrated).

But, in order to reason about the metrics one need to obtain them. In terms of the infrastructure, the execution environment will greatly influence the way one obtains metrics, in the sense that, if the application is running inside a virtual machine, the hypervisor can provide metrics about CPU utilization, incoming network requests, and so on. Running on containers is even better, because one can get those same metrics but with finer granularity, and the more isolated are the services the more service specific metrics can be obtained. As for the applications there are two possible approaches to be taken into account when trying to obtain metrics, the black box and the white box approaces. In the black box approach the application code is immutable, it can't be changed nor accessed, thus one cannot insert snippets of code to measure anything, hence containers need to be used, that define a perimeter around the application and measurements are taken around that perimeter. In the white box setting the application code is accessible, and one can insert probes in the code that allow to extract the desired metrics, as in case of Java applications where one can inject bytecode dynamically into the code.

Next we will present concrete techniques that can be used to monitor (distributed) applications and application components.

2.4.1 Monitoring Techniques

In this section we will provide some relevant examples of monitoring tools and techniques, that allow to retrieve metrics from infrastructures (SNMP) and from applications (Prometheus).

SNMP

Simple Network Management Protocol (SNMP) [33] is an Internet Standard protocol which provides the framework for the definition and interchange of management information about managed devices over TCP/IP networks and for remotely modifying that information to change the devices behavior. SNMP operates in the application layer of the Internet protocol suite and uses a simple scheme to routinely poll network devices status information that are exposed as system variables. These variables are organized in hierarchies that are described in the Management Information Base (MIB) of the device that contains control information, such as execution parameters and performance metrics (such as availability and response time).

Three significant versions of SNMP have been developed and deployed over the years: SNMPv1 [56], the original version of the protocol, SNMPv2 [57] and SNMPv3 [58].

Despite being considered a standard, SNMP has some significant limitations: *i)* the protocol is highly inefficient as its a pooling-based protocol and it's unable to transfer items in a block, having to retrieve one piece of information at a time; *ii)* it provides weak security features as it only allows two levels of authentication that is password based, which are transferred over the network unencrypted; *iii)* the information base is difficult to navigate and varies a lot with the manufacturers; and *iv)* it provides very low level and tight metrics, too close to the hardware, which sometimes it's not always what is desired.

Prometheus

Prometheus [54] is a centralized open-source system that provides both monitoring and alerting functionalities, written in Go. It is used by applications to record real-time metrics. Prometheus provides a multi-dimensional time series database where the time series collection is performed using a pull model over HTTP. Additionally, a flexible query language (PromQL) is prpovided, to enable administrators to leverage this dimensionality and real-time alerting features. Although being a centralized solution, Prometheus allows the possibility of operating in hierarchical federation, where several Prometheus instances cooperate with each other in a very well defined hierarchy that resembles a tree, and a cross-service federation, where a Prometheus instance of one service cooperates with another service's Prometheus instance in order to enable alerting and queries against both service's datasets in an integrated way.

Prometheus allows the user to obtain four different metrics: *i)* counter, a cumulative metric that represents a single monotonically increasing counter that can be used to obtain the number of requests served, tasks completed, or errors; *ii)* gauge, similar to counter but can arbitrarily go up and down, typically used for measured values like temperatures or current memory usage, and to obtain the number of concurrent requests or number of currently running processes; *iii)* histogram, another cumulative metric that samples observations, such as request durations or response sizes, and counts them in configurable buckets. Also provides a sum of all observed values; and *iv)* summary, similar to histogram but it also calculates configurable quantiles over a sliding time window.

2.4.2 Life Cycle Management Systems

Another important type of systems for our goals in this work are systems that control the life cycle of application replicas by deciding when and where to replicate. Next we will present two different life cycle management systems: CAUS and ENORM.

CAUS - Custom AutoScaler

CAUS [34] is a recent management system that enables elasticity for particular containerized microservices. Considering stateless microservices in a queue-based workload, CAUS works by attaching each container instance as a listener to the same request queue where requests arrive for processing. Then predicative approaches combined with a hybrid method are used to control the scaling of microservice instances, according to the demand and each microservice computing capabilities. These predicative approaches are used to obtain models for the arrival rate of requests so that scaling-up and down takes place beforehand based on the capacity that a single container is able to sustain. The hybrid method on the other hand, consists of two parts: *i)* one part reacts to changes in the load intensity by scaling container instances depending on their processing capability, and *ii)* the other part manages additional containers as a buffer to handle unpredictable load changes.

The objective is to create elastic microservices that perfectly adapt to changes in demand with instantaneous action and provide only the necessary resources to fulfill the current demand.

ENORM - Edge Node Resource Management

ENORM [70] is a resource management framework prototyped for the fog computing three tier infrastructure, where the top tier corresponds to the cloud tier, the middle tier corresponds to the edge node tier, and the bottom tier corresponds to the user device tier.

On the top tier, a cloud server manager is deployed for each application server. This server manages the task of communicating with edge nodes to request computing services, deploy partitioned servers on those edge nodes, and receive updates from the edge nodes in order to update the global view of the application server on the cloud.

ENORM operation ensures that, whenever a user starts an application in multiple regions that are covered by edge nodes, a connection between the edge nodes and the respective application servers on the cloud is established. Then, a partitioned application server is deployed to the edge nodes together with a copy of localized data that is relevant to the user in that location. Similarly to fog computing, the goal is for the user do communicate with the servers deployed on the edge nodes in order to reduce latency and improve QoS. The global view is maintained on the cloud server that is updated by the edge nodes, and when edge nodes cannot provide computing services, either due to being overloaded, or when the edge nodes are no longer able to improve the QoS, the deployed server will be terminated and users will need to connect to the cloud server as in a cloud-centric execution model.

2.4.3 Information Dissemination

Once data has been acquired locally, a decision has to be made about if and where to send that monitoring data. To ensure scalability and efficiency, components should rely on localized decisions, meaning that the acquired monitoring data 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 monitoring 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 [22] 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 [39]. In practice gossip is done over overlay networks (which will be discussed later in the Section 2.5.3), and 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 [42], GoCast [67], and Plumtree [35]. We do not discuss these approaches in detail because they because they fundamentally use the techniques described above, and due to lack of space.

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 [29]. The standard system model for publish/subscribe interaction relies on an event notification service, 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. There are two systems that are standard and used for this purpose, RabbitMQ [55] and Apache Kafka [6].

2.4.4 Discussion

Extending the monitoring capacity towards the edge of the system requires acquiring information across a large number of devices, possibly located on distant geographical areas. Additionally, to efficiently move or replicate services the monitoring system needs to collect information about which data is necessary for processing requests besides the standard performance metrics such as latency, throughput, and resource consumption. The monitoring system may also need to pre-process and co-relate different metrics locally and then deliver relevant data to local components responsible for the dynamic management of application logic and data planes. It's important to note that the monitoring system cannot have a high overhead as otherwise, its operation could negatively affect the system that is being monitored.

2.5 Building Blocks

In order to effectively extend applications from the cloud to the edge, to facilitate the monitoring of applications, and the replication of the application logic and data planes, particular technologies can be taken advantage of, namely Microservice Architectures, Virtualization Techniques, Overlay Networks, and Discovery Services for Microservices. We now briefly discuss these potentially useful building blocks.

2.5.1 Microservice Architectures

Microservice Architectures (MSA) [24, 41] is an emerging paradigm to develop, deploy, execute, and manage applications that can be seen as an evolution of the Service Oriented Arquitecture (SOA) [53]. Already adopted by the streaming giant Netflix [69], 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 [31].

MSA can 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 each microservice has its own database, and we want to dynamically replicate databases, we only have to worry about a single database at a time, which makes the problem easier to manage; *ii)* we want to replicate the application logic with fine-grain granularity, if the application is already decomposed into microservices that are self-contained, this greatly facilitates replication across the edge nodes; and *iii)* microservices can be efficiently moved/replicated to where they might have the most impact on performance, since usually they are relatively small software components.

2.5.2 Virtualization

The edge infrastructure contains a high amount of highly heterogeneous devices, some with limited capacities, thus not all edge resources support the direct execution of (all) application components. As an attempt to mitigate this problem, one can resort to virtualization and containerization techniques.

Virtualization [60] introduces a software abstraction layer, called virtual machine monitor (VMM) or hypervisor, between the hardware and the operating system and applications running on top of it, in the sense that it provides both abstraction and isolation of lower level functionalities and the underlying hardware. Each virtual machine (VM) is presented as an instance of the physical machine that gives users an illusion of accessing the physical machine directly. It's an elegant and transparent way to enable time-sharing and resource-sharing of the highly expensive hardware, where each VM is a fully protected and isolated instance containing its own operative system. Nowadays, there are multiple virtualization tools and techniques that were surveyed by Nanda et al. [16], the interested reader is directed to this survey to a more axhaustive discussion of such technologies.

Containers, on the other hand, consist of a light weight operating system running inside the host system, that run instructions native to the physical machine CPU [26]. Due to this, containers eliminate the need for instruction level emulation or just in time compilation, providing savings in resource consumption and isolation, without the overhead of virtualization. A relevant example of this technology is Docker [23], which provides the ability to manage Linux containers as self contained images.

Because of their lightweight nature, containers are potentially key to realize our vision of the cloud/edge environment, as they allow to have a runtime environment compatible with the various types of devices available in the edge, while providing some monitoring tools, with low overhead.

2.5.3 Overlay Networks

An overlay network is defined as a network which is deployed on top of another network, and the links that compose an overlay network are independent of the underlying network links and topology [39]. 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. These neighbour sets are typically maintained by a distributed membership protocol, which is responsible for dealing with filiation dynamics (making sure that the information of a peer leaving or joining the network is correctly handled at the overlay level). Overlay networks can be divided in two main classes, taking into consideration the mechanisms used to maintain local neighbor sets, which in turn defines the overlay topology: *i)* structured overlay networks, composed of logical links between peers whose topology has known properties a-priory, a common example is the ring topology where the nodes are organized according to their identifiers. Relevant examples of this overlay network are Chord [66], Pastry [59], and Kademlia [47]; and *ii)* unstructured overlay networks, such as Cyclon [68] and HyParView [36], where peers establish random neighbouring relationships such that it is impossible to predict where a node will be positioned.

When the application logic is spread across the various edge nodes, there will need to be a way for the microservices to interact with each other and with the databases. To interconnect different components across different cloud/edge nodes, lightweight and robust overlay networks can be leveraged. Ideally, the overlay links could map the relations between the database instances and the flow patterns of the monitoring information, and, combined with some gossip mechanisms, could be used to efficiently disseminate information across the various components.

2.5.4 Discovery Services for Microservices

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: high availability and be replicated), such as Apache ZooKeeper [7], but the most relevant one is the Netflix's Eureka [50], which was purposely built to support service discovery for Neflix's microservices based web site.

2.5.5 Discussion

All of the building blocks mentioned above allow to simplify the construction of efficient and robust applications in the cloud/edge setting. Particularly, leveraging on microservices architectures will allow to split the application in small independent components that can be more easily replicated to where they are most needed. Additionally, they allow to provide an easier deployment of these components across the highly heterogeneous edge devices (by leveraging on virtualization techniques), and to disseminate the monitoring information in a more efficient way (by taking advantage of network overlays).

2.6 Summary

In this Chapter we discussed techniques of cloud and edge computing, replication of data and application logic, monitoring, and some building blocks. 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

Proposed Work

Now that we have studied several existing standards and solutions, in this Chapter we will detail the objectives of the proposed work (Section 3.1), describe the use case application that is going to be leveraged during the course of this work (Section 3.2) and explain what is our initial approach to design and build Triforce (Section 3.3). Next, we will discuss the evaluation (Section 3.4) and work (Section 3.5) plan, and will finalize this document with brief final remarks (Section 3.6).

3.1 Refined Goals

The goal of this thesis is to develop a middleware solution that supports the automatic execution and life cycle management of applications based on microservice architectures, deployed across several geographic locations (in both cloud and edge infrastructures). The purpose of this middleware is to try to balance objectives that are somewhat conflicting: reduce user-perceived latency whenever possible, avoid server overload, and simultaneously reduce operational cost. These objectives are conflicting because one way to get low latency, dependability and availability, is to replicate everything everywhere, which leads to services becoming inefficient because of the overhead caused by replication protocols, which in turn leads to an increase in latency, and a lot of money being spent for the resources required to execute such a large number of components. If, on the other hand, we try to reduce the number of replicated components to the minimum, the user-perceived latency will be potentially very high as the servers will start to get overloaded. There must be a balance that has to be obtained during the distributed execution of the application, in a reactive way. If we perceive that users in some location are experiencing high latency, we will try to do something to minimize this, but in a very precise way.

In order to achieve this and to guide the decision making process, several metrics have to be taken into account: *i)* latency associated with throughput, which are indicators regarding the quality of service experienced by users; *ii)* operational cost, which can be measured considering how many resources are running at each moment; and *iii)* maximum load that can be imposed on servers, which will affect latency as overloaded machines will tend to take longer to process requests.

3.2 Use Case Application

The use case application that will be used in the development of our prototype is eShopOnContainers [28], an open source microservices and containers based application, that simulates an online shop. This reference application supports cross-platform execution at the server and client side as it was developed with .NET Core services, that are capable of running on Linux or Windows containers, and with Xamarin, that supports mobile apps running on Android, iOS, or Windows/UWP.

The application architecture, summarized in Figure 3.1, is a microservice oriented architecture implementation with multiple autonomous microservices: identity microservice, catalog microservice, ordering microservice, basket microservice, marketing microservice, and locations microservice. Each microservice has its own database, and each one is implemented with a different approach (simple CRUD vs. DDD/CQRS patterns) using Http as the communication protocol between the client apps and the microservices, with support to asynchronous communication for data updates propagation across multiple services based on Integration Events and an Event Bus (RabbitMQ or Azure Service Bus).

The application also provides three clients to consume the microservices: a MVC application developed with ASP.NET Core, a single page application developed with Angular and Typescript, and a Xamarin Mobile App.

This use case application will be leveraged in the evaluation process, as we will apply our solution to this concrete application and try to understand what is the impact this has on the application's performance, when variying the load imposed over the applications.

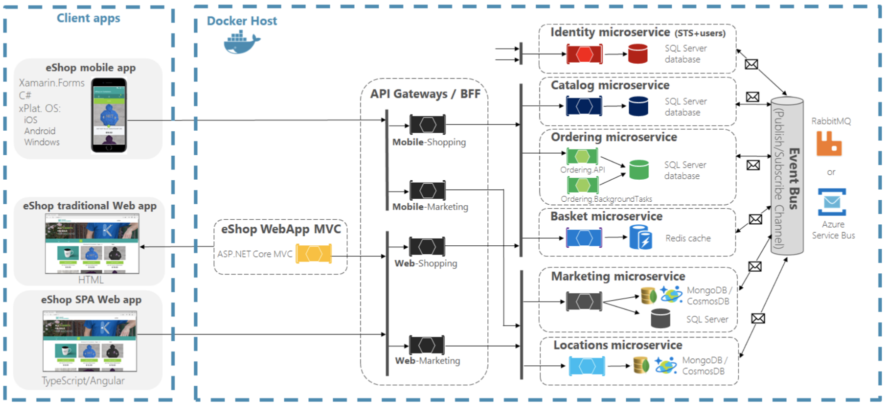


Figure 3.1: eShopOnContainers architecture - Extracted from [28]

3.3 Triforce Proposed Design

Enriching Microservice Architectures in order to become autonomic is the end goal of our middleware, where microservice-based applications adapt dynamically (on demand), taking advantage of the hybrid cloud/edge infrastructures. In order to make this possible, it is essential to build mechanisms that allow the autonomic control of the deployment, life-cycle, and binding of applications components, taking into account both the application logic and data planes. Thus four concrete problems that need to be addressed can be identified: *i)* management of application logic; *ii)* management of application data; *iii)* monitoring, that includes information dissemination; and *iv)* interaction between the various components.

To provide the desired performance, microservices should be executed in the most appropriate resource, taking advantage of the cloud/edge infrastructure. Deciding the ideal location however, is not an easy matter since it depends on multiple factors such as the microservice access pattern, its resource consumption, its interaction pattern and dependencies with other services, and the availability of computational resources. To address this challenge, we will add component that manages the application logic, that will consider these aspects and dynamically adjust the deployment of the microservices, and in essence perform three distinct tasks: migrate existing instances of microservices to a new more appropriate resource, create new instances and remove existing ones according to demand. To allow the execution of different microservices across the various edge resources as not all devices can provide the same execution environment, we will use lightweight containers to encapsulate the microservices.

Similarly to how the application logic is replicated, corresponding data should also be replicated with the service, to ensure the continuous and efficient operation of microservices that frequently manipulate state, as we want to avoid going to the cloud as much as possible. Thus, we will also add component that manages the application data that, similarly to the logic management component, is responsible for controlling the replication and deployment of data replicas, and essentially performs three tasks: migrate existing instances, create new database instances, and decommission existing ones. Replicating a full database however, can be highly inefficient and even impossible, as the particular edge resource where the replica is being deployed to may not have the computational and storage capabilities to host it. Thus, it is important to partition the database (according to parameters such as geographic location) and replicate only the necessary fraction that is actually being used. Additionally, additional replication steps might need to be employed in order to guarantee that the replicas converge to the same state, and the user doesn't see different versions of the application state when contacting different replicas.

In both cases discussed above, having a centralized master node that controls all the decisions is highly inefficient due to the large amount of resources available, thus we suggest to approach this problem by leveraging on local management decisions, that only consider resources and services in a close vicinity to the management node.

In order to make all this decision making possible, a monitoring component needs to be implemented that is able to acquire all the information necessary for the correct operation of the other two components, such as which data is necessary to replicate, what is the available capacity of the resources being used, what is the current latency on particular microservices, how many request are being processed, and so on. This information can be obtained leveraging on existing monitoring solutions such as Prometheus and on the information provided by the containers themselves.

Additionally, is it also necessary to propagate this information to local components that are responsible for the dynamic management of application logic and data planes. This can be done by leveraging on gossip based protocols combined with an overlay network. It is important to note that, similarly to the two components described above, we only want to propagate this information in a small vicinity to leverage on local decisions and avoid having a centralized node, as it is highly inefficient and network partitions can be fatal.

Finally, some microservices have dependencies among each other and need to communicate with each ohter. Thus there is a need to find a way to allow the microservices instances to select the nearest appropriate instance of other microservices it needs to interact with (ensuring locality). There exists a basic service called Eureka, produced by Netflix (discussed in Section 2.5.4) that can be leveraged for this purpose. A starting point would be to look at how Eureka is implemented and adapt it in a way to replicate this information on a local horizon and modify the resolving process to prioritize services that are closest to the invoker.

The proposed architecture of the middleware is schematized in the Figure 3.2.

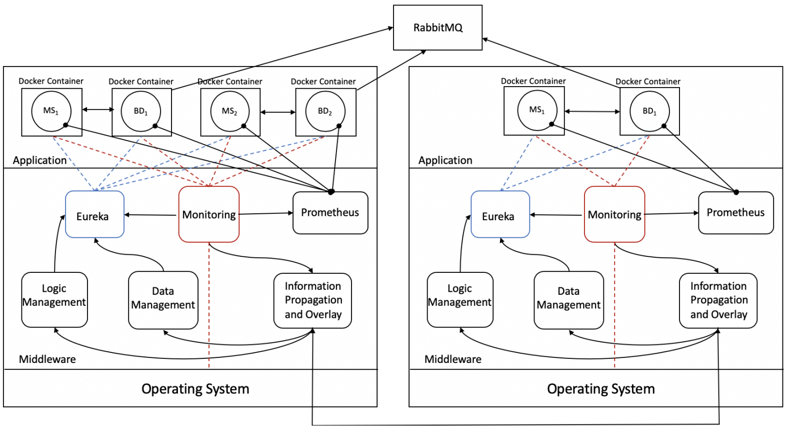


Figure 3.2: Proposed middleware solution architecture

3.4 Evaluation

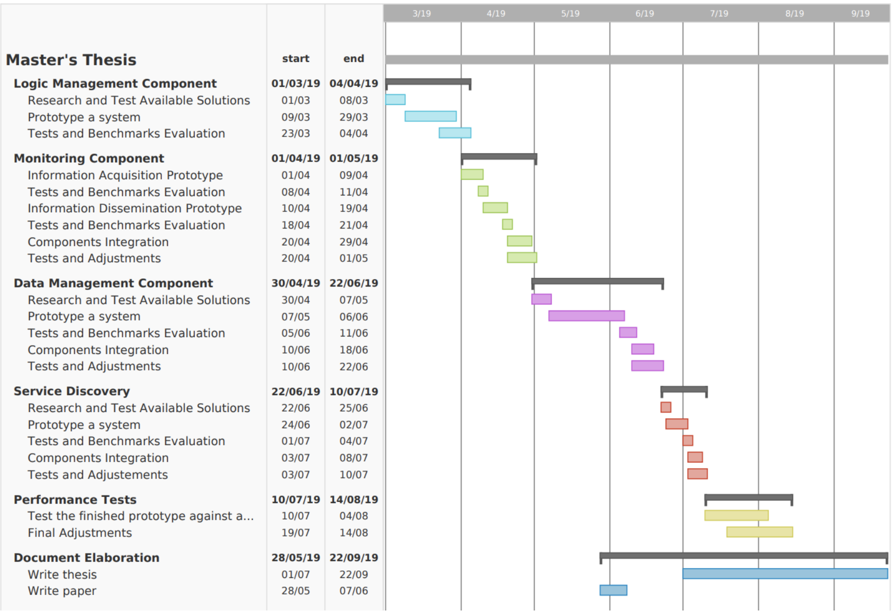
To evaluate our work, we will build a prototype of the proposed solution and run benchmark tests to compare it to a standard solution, where everything is running in a single or geo-replicated cloud datacenter, and we will measure what is the obtained benefit in terms of latency and throughput, and what is the disadvantage in terms of operation costs. In order to do that, both settings will be tested leveraging the use case application presented previously.

We plan to evaluate the two solutions using the previously described metrics: latency, throughput, server load, and operation cost. To obtain this metrics we will use available online tools such as JMeter [5], that allows to perform load and performance tests on static and dynamic applications by simulating a heavy load on the servers.

Throughout the development of the prototype, we will also perform several mini benchmark tests to guarantee that the solution is evolving with the expected performance.

3.5 Planning

In order to achieve our objectives, we will organize our work according to the following six main tasks and respective subtasks, that are presented in Figure 3.3.



Master's Thesis	start	end
Logic Management Component	**01/03/19**	**04/04/19**
Research and Test Available Solutions	01/03	08/03
Prototype a system	09/03	29/03
Tests and Benchmarks Evaluation	23/03	04/04
Monitoring Component	**01/04/19**	**01/05/19**
Information Acquisition Prototype	01/04	09/04
Tests and Benchmarks Evaluation	08/04	11/04
Information Dissemination Prototype	10/04	19/04
Tests and Benchmarks Evaluation	18/04	21/04
Components Integration	20/04	29/04
Tests and Adjustments	20/04	01/05
Data Management Component	**30/04/19**	**22/06/19**
Research and Test Available Solutions	30/04	07/05
Prototype a system	07/05	06/06
Tests and Benchmarks Evaluation	05/06	11/06
Components Integration	10/06	18/06
Tests and Adjustments	10/06	22/06
Service Discovery	**22/06/19**	**10/07/19**
Research and Test Available Solutions	22/06	25/06
Prototype a system	24/06	02/07
Tests and Benchmarks Evaluation	01/07	04/07
Components Integration	03/07	08/07
Tests and Adjustements	03/07	10/07
Performance Tests	**10/07/19**	**14/08/19**
Test the finished prototype against a...	10/07	04/08
Final Adjustments	19/07	14/08
Document Elaboration	**28/05/19**	**22/09/19**
Write thesis	01/07	22/09
Write paper	28/05	07/06

Figure 3.3: Work Plan

It is important to note that we will simultaneously design a solution and develop the prototype of the solution. In the end of the development of each prototype, tests and benchmark evaluation will be performed to ensure that the prototype is delivering the expected results. Also, at each stage, when we prototype and validate a solution, we will try to integrate it with the previously developed solution component to ensure that in the end we do not have much integration effort, and the designed solutions are compatible with each other.

Depending on the course of the work, we would like to submit a scientific publication to report our findings. At an initial level we would like to submit to the national computer conference INForum. Ideally, we would like to submit an extended version of this paper later for an international conference to be decided in the future (the work plan will need to be adjusted accordingly).

We realize that this project is ambitious and there may not be enough time to do everything. If this happens, some features will be selected and discontinued.

3.6 Final Remarks

In this document we presented the project to develop Triforce, a middleware support to enable dynamic microservices in the edge, whose objectives are to reduce user-perceived latency, to avoid the overload of servers and to reduce the operation cost. We plan to address this project in an incremental way, in which we first tackle the problem of the management of replication of microservices, then the problem of monitoring and information dissemination, the problem of decision making, the problem of management of data replication, and finally the problem of service discovery and interaction between microservices.

We will use as use case eShopOnContainers, an open source microservices and containers based application, and we will develop prototypes using this application as a guide, to ensure that the prototype supports the use case, and then, in the future, could be generalized. Finally, we will evaluate our solution by comparing it with a regular cloud deployment using the same use case.

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 ZooKeeper*. https://zookeeper.apache.org/. Accessed: 18-02-2019.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] S. N.T.-c. Chiueh and S. Brook. “A survey on virtualization technologies.” In: *Rpe Report* 142 (2005).

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] *Docker*. https://www.docker.com/. Accessed: 17-02-2019.

[24] 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.

[25] 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.

[26] R. Dua, A. R. Raja, and D. Kakadia. "Virtualization vs containerization to support paas." In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 610–614.

[27] 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.

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

[29] 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.

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

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

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

[33] R. Hunt. "SNMP, SNMPv2 and CMIP—the technologies for multivendor network management." In: *Computer Communications* 20.2 (1997), pp. 73–88.

[34] F. Klinaku, M. Frank, and S. Becker. "CAUS: An Elasticity Controller for a Containerized Microservice." In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: ACM, 2018, pp. 93–98. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186296. URL: http://doi.acm.org/10.1145/3185768.3186296.

[35] 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.

[36] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE. 2007, pp. 419–429.

[37] 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.

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] P. Maymounkov and D. Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric." In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[48] 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.

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

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

[51] 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.

[52] 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.

[53] 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.

[54] *Prometheus*. https://prometheus.io/. Accessed: 16-02-2019.

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

[56] *RFC 1155*. https://tools.ietf.org/html/rfc1155. Accessed: 16-02-2019.

[57] *RFC 1901*. https://tools.ietf.org/html/rfc1901. Accessed: 16-02-2019.

[58] *RFC 2570*. https://tools.ietf.org/html/rfc2570. Accessed: 16-02-2019.

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

[60] J. Sahoo, S. Mohapatra, and R. Lath. “Virtualization: A survey on concepts, taxonomy and associated security issues.” In: *2010 Second International Conference on Computer and Network Technology*. IEEE. 2010, pp. 222–226.

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

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

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

[69] 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.

[70] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. "ENORM: A Framework For Edge NOde Resource Management." In: *CoRR* abs/1709.04061 (2017). arXiv: 1709.04061. URL: http://arxiv.org/abs/1709.04061.

[71] 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.

[72] 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.

[73] 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.