André Vasconcelos Carrusca

Licenciatura em Engenharia Informática

Gestão de micro-serviços na Cloud e Edge

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

Orientadora: Maria Cecília Farias Lorga Gomes, Prof. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-orientador: João Carlos Antunes Leitão, Prof. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Fevereiro, 2018

Resumo

O aumento do número de dispositivos móveis nos últimos anos tem elevado o número de pedidos realizados aos serviços de *backend* da *cloud*, bem como a quantidade de dados que são produzidos. Este facto tem obrigado a que sejam utilizadas novas arquiteturas no desenvolvimento dos sistemas e encontradas novas estratégias de forma a garantir a qualidade dos serviços.

A arquitetura de micro-serviços, na linha de "*Service Oriented Architecture and Computing*" (SOA/SOC), permite o desenvolvimento independente de serviços de menor dimensão e desacoplado, implementando cada um uma funcionalidade particular, com uma interface bem definida e acessível através da rede. Serviços com funcionalidades mais complexas resultam da comunicação por mensagens entre os micro-serviços, em que cada um recorre aos serviços de outros. Esta arquitetura permite o *deployment* independente de cada serviço com configuração individual dos recursos (ex.: CPU, RAM), bem como o seu escalonamento independente (múltiplas instâncias por serviço) e, de acordo com as necessidades. O tamanho reduzido de cada micro-serviço permite também o seu *deployment* em arquiteturas heterogéneas de computação, como a *cloud* e a *edge*.

Devido à heterogeneidade dos locais de *deployment* considerados, ou seja, a *cloud* e *edge*, sendo que na *edge* ainda existe a questão dos recursos computacionais serem mais limitados, há uma grande complexidade em como a gestão dos micro-serviços tem de ser realizada, em particular na migração dos serviços. Em primeiro lugar existe o problema de decisão na migração um dado serviço, e para que local, tendo com base certas métricas (ex.: tempos de resposta), sendo que em seguida é necessário decidir como se processa essa migração. Ao existirem vários micro-serviços distintos, em que podem haver dependências entre eles, a gestão torna-se mais complexa pois é necessário decidir, aquando de uma migração de um serviço particular, se as suas dependências (outros serviços) também terão de ser migradas.

A solução consiste em desenvolver uma arquitetura com mecanismos automáticos de migração de micro-serviços entre a *cloud* e a *edge*, tal que permita uma diminuição no tempo de acesso a esses serviços, e que permita um melhor desempenho aplicacional, resultando numa redução nos custos. Estes mecanismos de migração de serviços irão permitir o *deployment* de micro-serviços de forma automática na *cloud* e *edge* consoante certas regras e métricas configuráveis (por exemplo, tempo de latência) e com base em dados fornecidos por um outro componente de monitorização.

Palavras-chave: Micro-serviços, *cloud*, *edge*, *deployment* de micro-serviços

Abstract

The growth of mobile devices number in recent years has increased the number of requests made to cloud backend services as well as the amount of data that is produced. This has forced the use of new architectures in the development of systems and to develop new strategies to guarantee the quality of services.

The microservice architecture, following the paradigm of "Service Oriented Architecture and Computing" (SOA/SOC), allows the independent development of smaller and decoupled services, each implementing a particular functionality, with a well-defined interface and accessible through the network. Services with more complex features result from the communication of messages between microservices, each one using the services of the others. This architecture allows the independent deployment of each service with an individual configuration of the resources (e.g., CPU, RAM), as well as its independent scheduling (multiple instances per service), according to the needs. The reduced size of each microservice also allows its deployment in heterogeneous computing architectures formed by cloud and edge nodes.

Due to the heterogeneity of the considered deployment sites, i.e., cloud and edge (with more limited resources), the system for the management and migration of microservices is quite complex. First, it is necessary to decide when to migrate a given service and to where, based on certain metrics (e.g., response times), and then it is necessary to decide how this migration is processed. When several distinct microservices are interdependent, the decision on if and how to migrate a service may implicate the migration of other services.

The goal of this work is to develop an architecture providing automatic mechanisms for microservices migration between the cloud and edge and its implementation. The resulting implementation aims to reduce the access time to these services with consequent better application performance and cost reduction. These service migration mechanisms will allow the deployment of microservices automatically in cloud and edge according to certain rules and configurable metrics (e.g., latency time) based on information provided by another monitoring component.

Keywords: Microservices, cloud, edge, microservice deployment

Índice

Lista de Figuras

Lista de Tabelas

Glossário

consistência eventual — É um tipo de consistência em que podem existir conflitos, isto é, para uma dada consulta podem ser apresentados diferentes resultados, mas existe uma comunicação entre os processos para tentar resolvê-los, para que a uma dada altura seja acordado o valor definitivo. Assim se durante um período de tempo, caso não existam mais alterações nos dados, todos os processos irão manter o mesmo valor para os dados, para quando exista uma consulta, eventualmente (*eventually*), seja apresentado o mesmo valor para todos os que consultarem [56].

consistência forte — É um tipo de consistência que assegura que apenas um estado consistente pode ser visto. Todos os processos do sistema mantêm um estado semelhante e devem retornar o mesmo valor para uma dada consulta [56].

Siglas

API	*Application Programming Interface*.
CQRS	*Command Query Responsability Separation*.
HTTP	*Hypertext Transfer Protocol*.
IaaS	*Infrastructure as a Service*.
IPC	*Inter-Process Communication*.
JSON	*JavaScript Object Notation*.
PaaS	*Platform as a Service*.
REST	*Representational State Transfer*.
SaaS	*Software as a Service*.
SOA	*Service-Oriented Architecture*.
SOAP	*Simple Object Access Protocol*.
SOC	*Service-Oriented Computing*.
URI	*Uniform Resource Identifier*.
VM	*Virtual Machine*.

Capítulo 1

Introdução

O presente capítulo tem como objetivo apresentar o contexto geral no qual se enquadra o trabalho, indicar o problema encontrado e fornecer uma breve explicação sobre a solução proposta, de forma a tentar solucionar o problema. Serão ainda apresentadas as contribuições esperadas que a solução venha a fornecer, bem como a estrutura geral do restante documento.

1.1 Contexto

A evolução da tecnologia nos últimos anos e a exigência por parte das pessoas pela inovação contínua, obrigou as empresas a ter que repensar a gestão das suas aplicações. Além de ser necessário lançar atualizações para corrigir erros, as empresas têm que estar em constante progressão de forma a fornecer novas funcionalidades aos seus utilizadores, pois sem isso ao longo do tempo vão perdendo os seus clientes, traduzindo-se numa possível perda de receitas. Outro aspeto que tem forçado a evolução dos sistemas, é o aumento do número de dispositivos móveis que se tem observado nos últimos tempos. Isto significa que, por parte destes dispositivos, existe um acréscimo na quantidade de dados produzidos e também no número de pedidos realizados aos serviços de *backend*. Devido a este facto, é necessário encontrar estratégias de forma a garantir a Qualidade de Serviço (QoS), pois se esta se degradar, por exemplo, se os tempos de resposta se tornarem elevados, os utilizadores tendem a procurar outros serviços.

As soluções arquiteturais mais tradicionais de desenvolvimento e *deployment* de aplicações não são as mais indicadas para este ritmo de inovação e melhorias. Para adicionar novas funcionalidades e fazer o *deployment* de uma nova versão de uma dada aplicação como, por exemplo, com uma arquitetura monolítica, tal não é possível de realizar da forma mais eficiente. Esta ineficiência está relacionada com a necessidade de fazer um *deploy* de uma nova versão integral da aplicação, não sendo possível realizar o *deploy* apenas do componente modificado, o que implica custos adicionais.

Para tentar solucionar estes problemas e introduzir novas funcionalidades como, por exemplo, a possibilidade de escalar apenas certos componentes de forma mais eficiente (sem que seja necessário escalar a aplicação toda), surgiu o conceito de micro-serviços [13, 16], uma nova arquitetura

aplicacional que segue a arquitetura/computação orientada a serviços (SOA/SOC, secção 2.1).

Um micro-serviço consiste num processo independente que implementa uma dada funcionalidade, em que este pode comunicar com outros micro-serviços, por mensagens através da rede, de forma a fornecer funcionalidades mais complexas. Uma aplicação composta por micro-serviços permite corrigir um problema localizado num dado micro-serviço, sem que para isso toda a aplicação fique indisponível. Isto facilita também a adição de novas funcionalidades, pois apenas é necessário desenvolver e realizar o *deploy* de um novo micro-serviço sem que existam conflitos com os já presentes. Com uma arquitetura de micro-serviços, o *deploy* de cada serviço é realizado de forma independente, permitindo que a configuração dos recursos (ex.: CPU, RAM) seja realizada por serviço, existindo também a possibilidade da existência de múltiplas instâncias por serviço, se tal for necessário. Os micro-serviços são uma arquitetura *cloud-native*, o que significa que a *cloud* é o local de *deployment* preferencial.

A computação na *Cloud* é um modelo de computação que permite o acesso partilhado a um conjunto de recursos como, por exemplo, armazenamento e capacidade de processamento, fornecidos como serviços. Tem ainda como vantagem, o aumento e a diminuição dos recursos alocados a uma aplicação de forma dinâmica, com base em certas métricas ou regras (por exemplo, a taxa média de utilização do CPU superior a 50%).

Devido ao tamanho reduzido de cada micro-serviço, é também possível, aproveitar outros tipos de locais para realizar a computação, como a *edge*. A computação na *Edge* é um paradigma que permite que a computação seja realizada em dispositivos mais próximos do utilizador, libertando assim a infraestrutura da *cloud* de certos processamentos. É então possível migrar certos serviços da *cloud* para a *edge*, permitindo uma diminuição no tempo de acesso a esses serviços, traduzindo-se num melhor desempenho aplicacional, podendo ainda proporcionar uma redução nos custos. A migração dos micro-serviços para a *edge*, permite também uma redução do volume dos dados transferidos entre os utilizadores e a *cloud*, fazendo primeiro uma agregação dos dados.

Devido à possibilidade da existência de bastantes micro-serviços por aplicação e também de diversos locais de *deployment* (*cloud* e múltiplos nós na *edge*), existe uma grande complexidade na gestão e na monitorização dos serviços.

1.2 Problema

A grande complexidade na gestão de aplicações compostas por micro-serviços em execução na *cloud* depende do número de micro-serviços que constituem uma aplicação bem como do número de utilizadores dos diferentes serviços e QoS (Qualidade de Serviço) esperada, implicando alterações na escalabilidade dos diferentes serviços. Numa aplicação que exija tempos de latência baixos das operações realizadas pelos utilizadores, em que estes poderão aceder aos serviços através de dispositivos móveis (*smartphones*, *tablets*), *wearables* (*smartwatches*), a possibilidade de ter serviços disponíveis na *edge* permite que a comunicação entre os dispositivos e o serviço seja mais rápida, bem como a filtragem de dados, diminuindo o volume dos dados a enviar para a *cloud*.

Embora a migração dos serviços para a *edge* apresente diversas vantagens, torna, no entanto, necessário lidar com o problema da existência de um maior número possível de locais de *deployment*, bem como com a sua heterogeneidade. As decisões têm de ter em atenção os recursos desses nós, a sua localização face aos pedidos de acesso, o volume de dados em trânsito, assim como a possível replicação dos dados de um serviço (ex.: entre um datacenter na *cloud* e um nó na *edge*). No contexto particular deste trabalho, pretende-se abordar os problemas relacionados com a gestão dos

micro-serviços, concretamente em relação à migração de serviços da *cloud* para a *edge*. O problema será abordado de uma forma incremental, subdividindo-se nos seguintes sub-problemas:

1. Migração de um único micro-serviço de forma dinâmica (em tempo de execução), considerando a inexistência de dependências em relação a outros micro-serviços. Avaliação de quais as métricas adequadas à decisão sobre essa migração, tendo em atenção a heterogeneidade dos recursos, as características do serviço, e a qualidade de serviço que se quer garantir.
2. Migração, em tempo de execução, de um micro-serviço que possa depender de outros micro-serviços. A decisão é mais complexa porque pode implicar a migração de um conjunto de serviços, em vez de apenas um único. Avaliação de quais as métricas adequadas à decisão sobre essa migração, adicionando os resultados obtidos no ponto 1.
3. Automatização do processo de migração, com base nos resultados obtidos em 1 e 2. Definição de uma arquitetura que permita a recolha atualizada de dados necessários à decisão, bem como a codificação necessária de regras e métricas que permitam que o processo de migração seja automático, ou pelo menos com uma menor intervenção humana. Esta gestão automática tem de ser eficiente, não comprometendo a qualidade da experiência final dos utilizadores, nem o custo envolvido.

O objetivo geral da gestão automática da migração de micro-serviços é possibilitar o melhor desempenho de uma aplicação ao reduzir a comunicação com a *cloud*, melhorando os tempos de resposta através de componentes localizados na *edge*, e reduzindo o volume de dados a transferir. Este objetivo é bastante importante para aplicações que requeiram uma baixa latência nas operações. Pode também permitir que os custos das infraestruturas na *cloud* sejam reduzidas, ao diminuir a dimensão das instâncias a replicar bem como o volume de dados a guardar.

1.3 Solução proposta

Esta dissertação consiste em desenvolver, para uma aplicação com uma arquitetura de micro-serviços, mecanismos de gestão automática que permitam a migração da *cloud* para *edge* (e vice-versa) dos micro-serviços que compõem a aplicação. A solução terá que ter em conta quais as métricas que contribuem para as decisões de migração, como características dos serviços, tempo de latência, volume de acessos/número de utilizadores e sua localização, tipo e custo dos recursos, etc. Terá, além disso, de abarcar o tipo de migração a ser efetuada em cada caso concreto, por exemplo, replicar serviços apenas no contexto da *cloud*, replicar serviços para a *edge*, ou executar exclusivamente esses serviços na *edge*.

Será também necessário identificar como a coordenação entre os serviços irá ser realizada, por exemplo, tendo duas instâncias de um dado serviço, uma na *cloud*, outra na *edge*, que têm repositórios de dados privados, que mecanismos serão necessários para existir uma consistência nos dados observados por cada réplica, sendo que esta questão poderá influenciar o tipo de migração a realizar.

A definição da arquitetura irá permitir a realização de uma avaliação mais exaustiva do estado da arte neste domínio, como, por exemplo, as dimensões referidas pela computação osmótica (secção 2.5). Este paradigma tem como objetivo permitir o *deployment* automático de micro-serviços tanto na *cloud* como na *edge*, e implica uma gestão dinâmica dos serviços de forma a que, a migração entre estes dois ambientes seja realizada sem degradar o desempenho da aplicação. A

Figura 1.1 apresenta uma primeira visão simplificada da arquitetura da solução e seus componentes. O elemento de Migração (‘*Migration*’) será o componente a ser desenvolvido no contexto desta dissertação e, tal como representado na figura, é responsável pela migração dos micro-serviços. Esta migração será efetuada com base nas informações fornecidas pelos elementos de Monitorização (‘*Monitoring*’), responsável pela obtenção de informação sobre os serviços em execução na *cloud* e na *edge*.

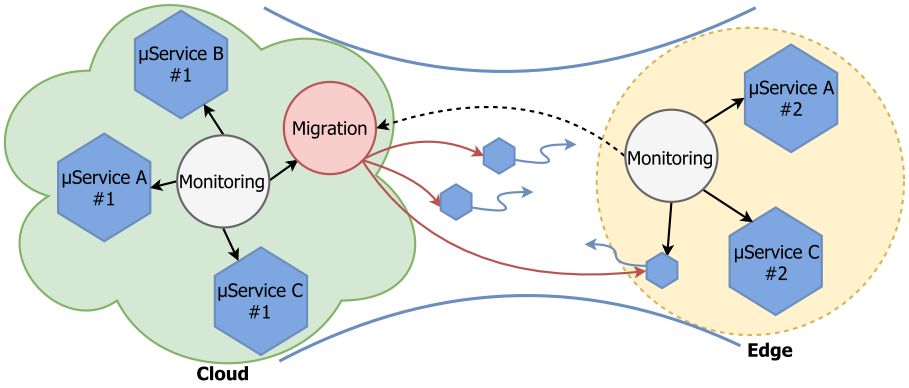

Figura 1.1: Solução proposta.

Tendo em vista a resolução dos sub-problemas identificados na secção 1.2, o trabalho começará por avaliar uma aplicação particular com uma arquitetura de micro-serviços, previamente desenvolvida, a *Sock Shop* [57], e que é um protótipo de uma aplicação web de comércio eletrónico de venda de meias. A *Sock Shop* foi escolhida tendo em conta que é *open-source* (outra possibilidade seria [7]), possui uma boa documentação e tem um nível de complexidade adequado para o problema que se pretende resolver.

1.4 Contribuições

O objetivo desta dissertação é implementar um sistema com uma arquitetura de micro-serviço que incorpore mecanismos de gestão e coordenação automáticos na migração dos serviços entre a *cloud* e a *edge*. No final da implementação são esperadas as seguintes contribuições:

1. Conceção de uma aplicação em micro-serviços que incorpore funcionalidades de gestão e coordenação automáticas dos micro-serviços, no que diz respeito à sua migração entre a *cloud* e a *edge*.
2. Avaliação da aplicação, de forma a verificar a viabilidade dos mecanismos desenvolvidos.

1.5 Estrutura do documento

Este documento, para além do presente capítulo, é composto por mais dois.

O capítulo 2 diz respeito ao estado da arte relacionado com os conceitos que a solução aborda. É explicado em detalhe o conceito de micro-serviços (secção 2.2), a sua arquitetura, as vantagens que apresenta, bem como os desafios associados. De seguida é abordada a computação em *Cloud* (secção 2.3), um tipo de computação fundamental para o *deploy* de micro-serviços, em que são tratadas as suas características principais, e também são fornecidos alguns exemplos de infraestruturas públicas. Depois é apresentada a computação na *Edge* (secção 2.4), um tipo de computação

que complementa a computação na *Cloud*, permitindo que certas operações sejam realizadas de forma mais próxima do utilizador. Em seguida, a secção 2.5 analisa a problemática da computação numa infraestrutura heterogénea composta pela computação na *Cloud* e computação na *Edge*, tendo em vista a existência de uma gestão automática do *deployment* de micro-serviços nos dois níveis. No final, é apresentado um resumo sobre os conteúdos apresentados no capítulo.

O capítulo 3 apresenta mais detalhes sobre a solução proposta, nomeadamente que ferramentas e software serão implementados, bem como as metodologias e ferramentas que se espera que venham a ser utilizadas para o seu desenvolvimento. No final é fornecido um plano de trabalhos para contextualizar a forma como se irá desenvolver o projeto.

Capítulo 2

Estado da arte

Este capítulo tem como objetivo a apresentação de conceitos, definições e abordagens relevantes para o desenvolvimento da solução proposta. A secção 2.2 aborda o conceito de micro-serviços, as vantagens e desvantagens da utilização, principais abordagens para o desenvolvimento de uma aplicação sob uma arquitetura de micro-serviços, como transitar de uma arquitetura mais clássica, como a monolítica, para esta arquitetura e alguns casos práticos. São ainda descritas algumas características de suporte à execução de micro-serviços no contexto de computação na *Cloud*, o qual é tratado com mais detalhe na secção seguinte. A secção 2.3 descreve as características principais da computação na *Cloud*, a gestão dos seus recursos, bem como alguns exemplos de infraestruturas públicas na *cloud*. A secção 2.4 refere-se à computação na *Edge*, explicando no que consiste, bem como apresenta as suas vantagens e que desafios se levantam na sua utilização. A secção 2.5 discute a problemática da computação numa infraestrutura heterogénea composta pela computação na *Cloud* e computação na *Edge*, e uma proposta recente neste domínio.

2.1 Computação orientada a serviços - SOC

A computação orientada a serviços (*Service-Oriented Computing* - SOC) é um paradigma de programação, que tem como base as propostas de design seguidas pela arquitetura orientada a serviços (*Service-Oriented Architecture* - SOA). A sua visão é obter uma infraestrutura flexível e disponibilizar como serviços, funcionalidades aplicacionais independentes e reutilizáveis, separando as funcionalidades em três camadas: na inferior encontram-se os serviços base, na camada intermédia a composição de serviços e no topo a gestão e monitorização de serviços [32].

Os serviços são componentes autónomos e *loosely coupled*, que são usados no paradigma SOC para possibilitar um desenvolvimento mais rápido e barato de aplicações distribuídas [32]. Este paradigma emergiu devido à necessidade de dar resposta à evolução de como as empresas pretendiam gerir os seus negócios - ao invés de existirem apenas aplicações ("serviços") desenvolvidas por uma única empresa, o paradigma de serviço permite que várias empresas desenvolvam as suas aplicações, e que estas comuniquem e interajam através de *Web services*. Um *Web service* é um tipo de serviço específico que é identificado por um *Uniform Resource Identifier* (URI) [31].

Existe a possibilidade de agregar vários serviços, denominada composição de serviços, permitindo que sejam criados novos serviços, e que pode ser alcançada através de duas abordagens:

- **Orquestração**: consiste na construção de um processo, em que este é responsável pela coordenação da interação entre os diferentes serviços, sendo incluído neste tipo de composição a gestão de transações e o tratamento de exceções.
- **Coreografia**: os serviços trocam mensagens entre eles, sendo que são definidas regras para a sua interação.

Este tipo de serviços pode ser desenvolvido utilizando duas soluções principais: *Web services* SOAP e *Web services* RESTful [49].

Os *Web services* SOAP são uma tecnologia mais tradicional, onde as interações entre os diferentes serviços são realizadas através de chamadas *Simple Object Access Protocol* (SOAP), em que o transporte dos dados é feito num formato XML. A descrição das suas interfaces é feita utilizando WSDL (*Web Services Definition Language*) e o UDDI (*Universal Description, Discovery, and Integration*) é um protocolo que contém um diretório onde estão as suas descrições, permitindo que os serviços sejam localizados e seja possível ver os seus detalhes.

A BPEL/BPEL4WS (*Business Process Execution Language for Web Services*) define uma linguagem para a composição de serviços numa forma de processos de negócio, em que cada composição é um processo de negócio ou um *workflow* que interage com um conjunto de *Web services* para atingir um determinado objetivo. A sua composição é denominada de processo e os serviços com o qual este interage são denominados de parceiros, sendo que um processo, como qualquer *Web service*, suporta um conjunto de interfaces WSDL que possibilita a troca de mensagens com os seus parceiros. A interação entre a composição BPEL e os seus parceiros é, no caso geral, *peer-to-peer*, em que cada parte invoca operações das interfaces públicas de cada um. Certas aplicações podem apenas utilizar um processo como um serviço sem fornecer qualquer funcionalidade, enquanto outros podem ser utilizados pelo processo como serviços [8].

Os *Web services* RESTful são uma tecnologia mais recente, que surgiu como uma alternativa mais simples e leve comparando com os *Web services* SOAP, e que foram idealizados para facilitar o seu acesso e a composição de vários serviços [17]. Os serviços seguem a arquitetura *Representational State Transfer* (REST), a qual fornece um conjunto de restrições arquiteturais que se focam na escalabilidade dos componentes, o seu *deployment* independente e na implementação de interfaces genéricas [11]. Este tipo de serviços usa os métodos (`PUT`, `POST`, `GET` e `DELETE`) de invocação remota HTTP que são aplicáveis a qualquer recurso através do seu URI [17]. De acordo com [17], existem 4 propriedades que estes serviços apresentam:

- Os recursos representam o estado da aplicação do lado do servidor (*server-side*).
- Cada recurso utiliza um URI único.
- Os recursos têm uma interface uniforme através dos métodos HTTP para interagirem com as aplicações.
- A interação com um recurso é *stateless* [1].

[1] *Stateless* significa que todos os pedidos devem ter a informação necessária para o seu processamento, e não pode ser armazenado qualquer contexto no servidor, fazendo com que todo o estado da sessão seja mantido no cliente [11].

Existem diversas vantagens em utilizar *Web services* RESTful em vez de *Web services* SOAP como, por exemplo:

1. Utilização de interfaces uniformes imutáveis, sendo que não existe o problema da quebra de acessos nos clientes [33]. 2. Protocolo suportado pela maior parte dos clientes, visto não ser necessário qualquer tipo de *middleware*, já que as invocações são feitas por HTTP [33]. 3. As mensagens SOAP são bastantes descritivas, produzindo maior tráfego na rede devido seu tamanho e causando uma maior latência quando comparado com *Web services* RESTful [28]. 4. A codificação e descodificação de mensagens SOAP consomem mais recursos, provocando *overheads* visíveis no seu desempenho [28].

Num teste de performance, realizado num ambiente de computação móvel que comparou *Web services* SOAP e *Web services* RESTful, foi possível concluir [28]:

- O tamanho das mensagens utilizadas por *Web services* RESTful são cerca de 9 a 10 vezes menores que em *Web services* SOAP.
- O processamento e transmissão das mensagens é 5 a 6 vezes mais rápido em *Web services* RESTful que em SOAP.

Concluindo, a computação orientada a serviços tem como objetivo o desenvolvimento de aplicações através da disponibilização de serviços, sendo estes *loosely coupled*, com o propósito de fornecer funcionalidades independentes. Este tipo de paradigma de computação foi uma das bases dos micro-serviços e na sua arquitetura, descrito em detalhe na secção 2.2.

2.2 Micro-serviços

O termo micro-serviços refere-se a um pequeno processo independente que implementa uma dada funcionalidade de forma completa, podendo funcionar autonomamente sem recurso a outro tipo de serviço. O estilo arquitetural de micro-serviços baseia-se no desenho de aplicações em que um conjunto de micro-serviços independentes "cooperam" entre si de forma a fornecer certas funcionalidades, interagindo através de mensagens enviadas pela rede. Sendo que cada micro-serviço é independente, isto permite que cada um deles possa ser desenvolvido num tipo de tecnologia específica, não existindo por isso qualquer tipo de limitação à linguagem de programação, plataforma de desenvolvimento ou *framework* que é utilizada [10, 13].

Para melhor compreensão deste estilo arquitetural e quais as suas vantagens bem como as suas desvantagens, é relevante explicar um estilo arquitetural mais tradicional, o monolítico. Uma aplicação monolítica é desenvolvida como uma unidade única e é organizada em módulos dependentes da aplicação. Estes módulos não podem assim ser reutilizados por outra aplicação [10]. Um exemplo clássico da utilização deste tipo de arquitetura são as aplicações (Web) compostas por uma interface de utilizador (constituída por ficheiros HTML e Javascript), uma base de dados e a aplicação/ lógica do lado do servidor. Quando é necessária ser realizada qualquer alteração, tem de ser feito o *deploy* completo da aplicação, mesmo que seja uma alteração mínima num componente específico [13]. Para além deste inconveniente, à medida que a aplicação vai evoluindo e crescendo, torna-se difícil manter uma boa estrutura modular, fazendo com que se torne complicado manter as alterações de um dado módulo apenas dentro deste sem afetar os outros.

Estas são algumas das desvantagens que um sistema desenvolvido sob o estilo monolítico pode apresentar, e é aqui que começou a ganhar mais interesse o estilo arquitetural de micro-serviços, de forma a tentar ultrapassar estas dificuldades entre outras.

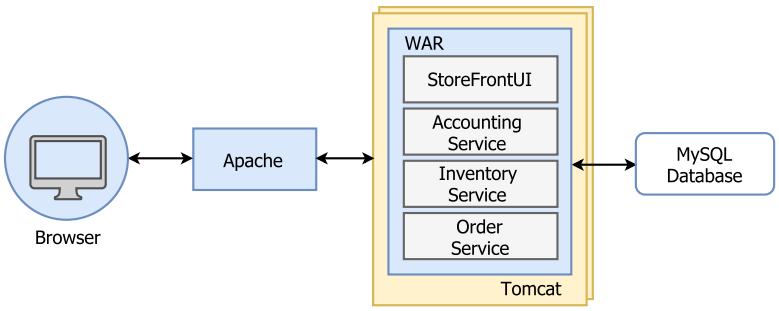

Figura 2.1: Diagrama de uma aplicação monolítica. Adaptado de [40].

A Figura 2.1 mostra quais os componentes de uma aplicação de comércio online que permite encomendas de clientes, verifica o inventário e o crédito disponível, e processa as encomendas. A aplicação é composta por vários componentes: 'StoreFrontUI' que implementa a interface, e os restantes componentes de *backend*.

A aplicação é implementada através de uma arquitetura monolítica, num único ficheiro WAR que corre num *web container* Tomcat, estando também conectada a uma única base de dados [40].

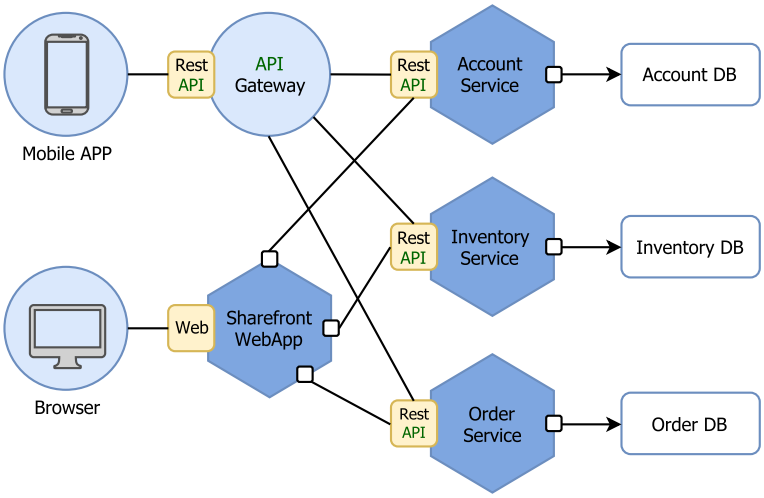

Figura 2.2: Diagrama de uma aplicação em micro-serviços. Adaptado de [39].

A Figura 2.2 representa a mesma aplicação, implementada em micro-serviços. A aplicação é composta por quatro micro-serviços que fornecem as funcionalidades da aplicação, em que o 'Sharefront WebApp' é de *frontend* e os restantes de *backend*, em que cada um destes tem uma base de dados privada. Existe ainda uma aplicação móvel que comunica com uma API Gateway (descrita em 2.2.3.2) que redireciona os pedidos para os respetivos serviços.

2.2.1 SOA e micro-serviços

Os micro-serviços podem ser vistos como uma evolução da arquitetura orientada a serviços (*Service-Oriented Architecture* - SOA), em que apesar de diferirem, partilham alguns aspetos [16].

A SOA surgiu como uma forma de mitigar os problemas e desafios das grandes aplicações monolíticas [29]. A sua visão é obter uma infraestrutura flexível e disponibilizar como serviços funcionalidades aplicacionais independentes e reutilizáveis [32]. Tem também como objetivo facilitar a manutenção do software, visto que é possível substituir um serviço por outro, se a semântica do novo serviço não mudar muito [29].

Uma das grandes diferenças entre a arquitetura de micro-serviços e a SOA, é a granularidade de cada serviço. Nos micro-serviços, os serviços fornecem apenas uma única funcionalidade, tendo por isso uma granularidade fina, enquanto que na SOA estes tendem a englobar uma funcionalidade do negócio, por vezes implementada como um subsistema completo. Outra diferença é que na SOA existe o conceito de maximizar a partilha de componentes, e numa arquitetura de micro-serviços, o objetivo é minimizar essa partilha [35].

Assim os micro-serviços emergiram da sua utilização real, tirando partido da melhor compreensão de sistemas e arquiteturas, como uma forma de melhorar a SOA [29].

2.2.2 Vantagens e desvantagens da arquitetura de micro-serviços

Em baixo são descritas algumas das vantagens da utilização da arquitetura de micro-serviços em relação a uma arquitetura mais tradicional, como a monolítica [14]:

- **Limites dos módulos bem definidos**: uma das características dos micro-serviços é a sua estrutura modular, um aspeto bastante importante num sistema em que trabalham equipas de grandes dimensões. A divisão do software em módulos é importante, pois em programas complexos, permite a realização de uma alteração, apenas com conhecimento mais detalhado de uma pequena parte do sistema, reduzindo a complexidade e o espaço de manutenção do software.

  Um sistema monolítico pode apresentar uma boa estrutura modular, mas é complexo de a manter. Por exemplo, caso haja necessidade de implementar rapidamente novas funcionalidades, pode acontecer que elas sejam implementadas de uma forma incorreta, pois é fácil contornar os limites de cada módulo, fazendo com que se prejudique no futuro a produtividade da equipa. Ao distribuir os módulos em micro-serviços separados, tornam-se os limites melhor definidos, impossibilitando encontrar formas de contornar este tipo de estrutura.

  A separação dos repositórios de dados por micro-serviço também é um aspeto importante para manter uma estrutura modular. Cada serviço terá a sua própria base de dados, e para obter os dados de outro serviço terá de ser feito com recurso às APIs dos mesmos.

- ***Deployment* independente**: os micro-serviços são componentes que permitem o seu *deployment* independente. Ou seja, quando é realizada alguma alteração, geralmente ela está confinada a um micro-serviço, pelo que apenas é necessário testar e fazer o seu *deployment*. Isto permite que o sistema não seja parado quando é feito um *deployment*, e depois deste ser realizado, caso exista algum problema num dado componente, este não deve fazer com que outras partes do sistema deixem de funcionar.

  É também possível obter escalabilidade de uma forma mais precisa e eficiente, i.e., ao nível do serviço. Isto é útil, pois nem todos os serviços que compõem a aplicação têm a mesma carga, isto é, podem existir serviços que os utilizadores necessitem mais e por isso são mais utilizados. Por outro lado, nem todos os serviços necessitam dos mesmos recursos (ex.: CPU,

RAM), possibilitando que um dado serviço seja capacitado de mais recursos, se assim se justificar, e outro, que não necessite de tantos recursos, seja diminuído de recursos.

- **Utilização de múltiplas tecnologias**: como cada um dos micro-serviços é uma unidade independente, é possível usar diferentes tecnologias (linguagens de programação, *frameworks* e repositórios de dados) em cada um deles, permitindo assim escolher a tecnologia que melhor preencher os requisitos pretendidos.

  Num sistema monolítico assim que são determinadas as linguagens e *frameworks* no qual este será implementado torna-se complicado modificar essa escolha, pois o custo da alteração num estado avançado de desenvolvimento seria demasiado elevado.

Apesar da existência de vantagens, existem também certos inconvenientes associados à utilização de micro-serviços [14]:

- **Distribuição**: uma aplicação de micro-serviços pode ser considerada um sistema distribuído, pois cada serviço é executado de forma independente, podendo residir em máquinas distintas. Porém, o software distribuído pode apresentar alguns inconvenientes.

  A performance é um aspeto bastante crítico, pois as chamadas remotas são lentas. Se um serviço utilizar um conjunto de chamadas a serviços remotos, em que cada um destes também chama serviços remotos, os tempos de respostas podem ser maiores, resultando numa maior latência. Ao fazer chamadas assíncronas em paralelo (onde as chamadas são independentes umas das outras) é possível minimizar este problema, tendo apenas influência a chamada mais lenta, ao invés da soma da latência de todas chamadas.

  A robustez também é um ponto importante, pois numa chamada remota pode existir uma falha em qualquer momento, e num sistema composto por centenas de micro-serviços existem ainda mais potenciais pontos de falha.

- **Consistência dos dados**: manter consistência forte é bastante complicado num sistema distribuído, por exemplo, a lógica de negócio pode fazer decisões baseadas em dados inconsistentes.

  Os micro-serviços introduzem problemas ao nível da consistência observada entre dados de diferentes micro-serviços (por exemplo, consistência eventual) pela sua forma de descentralizar a gestão dos dados, sendo por vezes necessário atualizar múltiplos recursos, aumentando a possibilidade de incoerência (por exemplo, porque um serviço não se encontrava disponível).

- **Complexidade operacional**: sem o auxílio de mecanismos de automatização e colaboração, é bastante complicado gerir e realizar o *deploy* de centenas de micro-serviços. A complexidade operacional também aumenta devido à necessidade de gestão e monitorização destes serviços.

  Apesar da utilização de micro-serviços tornar mais fácil a compreensão de cada componente, esta separação pode fazer com que seja complexo o *debug* de serviços que utilizam chamadas a outros serviços.

2.2.3 Arquitetura de micro-serviços

Para implementar um sistema com uma arquitetura de micro-serviços é preciso seguir certas diretrizes, desde o seu design, a forma como é implementado, o seu *deployment* e certos aspetos em tempo de execução.

Esta secção está organizada de acordo com a taxonomia representada na Figura 2.3, adaptada de [16], sendo que certos tópicos na figura, estando relacionados, serão abordados em conjunto. Certas abordagens descritas, como em termos de comunicação, gestão de dados, descoberta de serviços, entre outras, são um subconjunto baseado em [47].

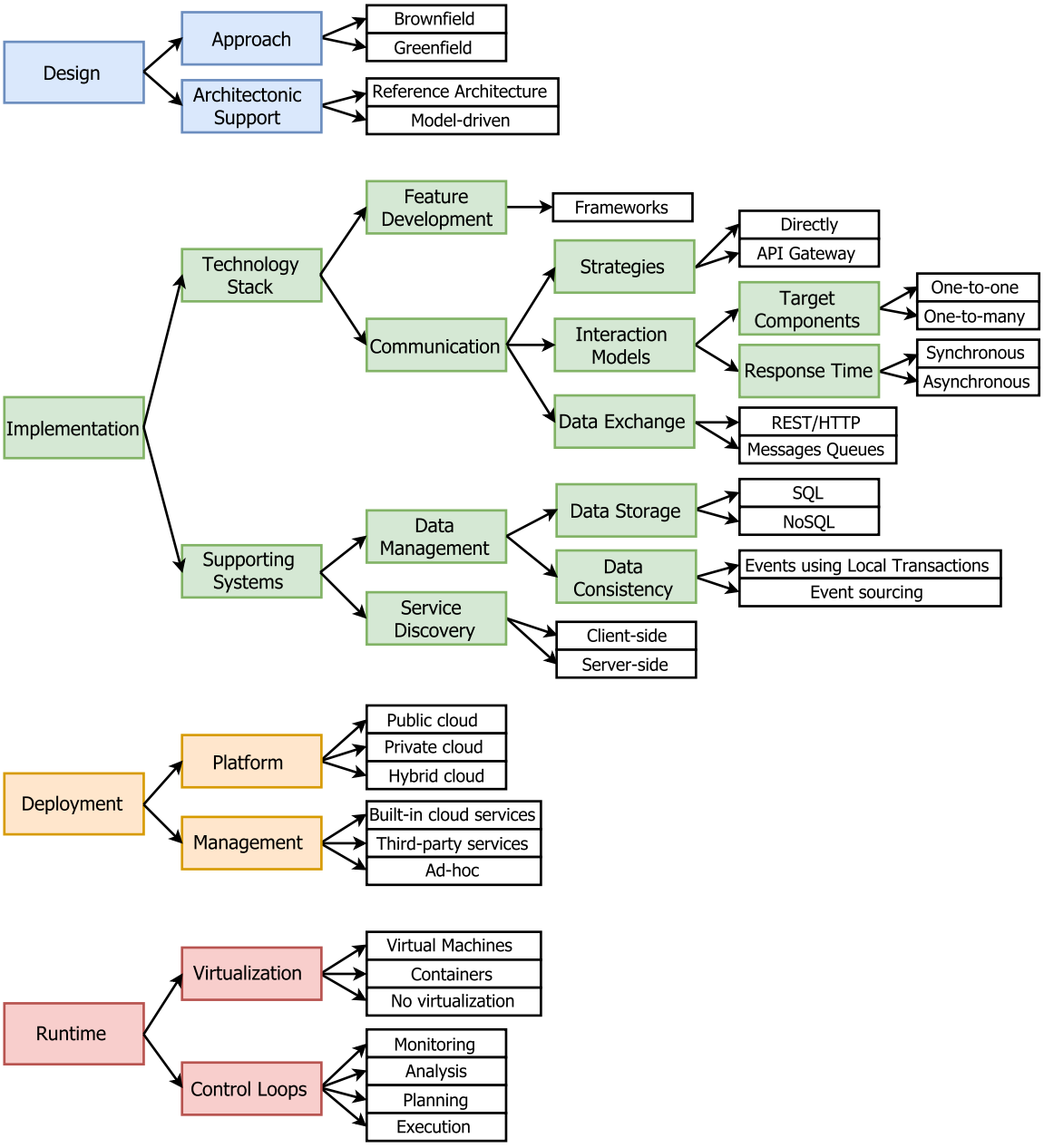

Figura 2.3: Taxonomia da arquitetura de micro-serviços. Adaptado de [16].

2.2.3.1 Design

Abordagem (*Approach*) No desenvolvimento de uma aplicação com uma arquitetura de micro-serviços, a existência (ou inexistência) de software antigo que deve transitar para esta arquitetura, influencia a abordagem inicial a ser seguida, podendo ser **Brownfield**, caso exista a necessidade de migrar software antigo, ou **Greenfield** caso não seja necessária a integração com software existente [16, 20].

Suporte Arquitetónico (*Architectonic Support*) A arquitetura de um sistema tem de ser idealizada de forma a cumprir as funcionalidades e restrições definidas, sendo que para isto é necessário seguir certas metodologias. Por exemplo, a utilização de uma **Arquitetura de Referência** ou a **Arquitetura orientada ao Modelo** (*Model-Driven Architecture* - MDA) podem auxiliar neste processo [16].

2.2.3.2 Implementação (*Implementation*)

Tecnologias (*Technology Stack*) As Tecnologias foram subdivididas em, Desenvolvimento de Funcionalidades (*Feature Development*), e em Comunicação (*Communication*). A primeira dimensão diz respeito ao desenvolvimento das funcionalidades principais dos micro-serviços, ou seja, qual o propósito do micro-serviço. A segunda dimensão refere-se às abordagens sobre como a comunicação com (e entre) os micro-serviços é realizada.

Desenvolvimento de Funcionalidades - *Frameworks* Um micro-serviço implementa um conjunto de funcionalidades e o seu desenvolvimento beneficia da utilização de uma *framework* que ofereça algumas funcionalidades base, permitindo assim acelerar o seu desenvolvimento. Segue-se a descrição de um conjunto popular de *frameworks* que auxiliam o desenvolvimento de micro-serviços, tendo como base a análise apresentada em [15].

- **Dropwizard**: fornece bibliotecas Java estáveis, disponíveis em pacotes simples e leves.

  A sua modularidade interna permite que a aplicação seja pequena, reduzindo o tempo de desenvolvimento e manutenção.

- **Light Rest 4J**: é uma *framework* de APIs REST construída com base na light-4j, também uma *framework* de micro-serviços rápida, leve e nativa. O Light Rest 4J é umas das *frameworks* Java mais rápidas de micro-serviços [34].

- **Spring Boot**: facilita a criação de aplicações e serviços de forma bastante organizada.

Na Tabela 2.1 é possível analisar alguns prós e contras das *frameworks* identificadas.

Comunicação - Estratégias (*Communication - Strategies*) Por vezes, existe a necessidade de um micro-serviço comunicar com outros. Para permitir esta comunicação é necessário expor APIs, como por exemplo, uma API REST, que podem depois ser invocadas por um outro micro-serviço. As APIs expostas podem também ser utilizadas por aplicações, por exemplo, uma aplicação mobile, ou por outros serviços externos, sendo que existem duas abordagens possíveis para a invocação dos micro-serviços:

Tabela 2.1: Prós e contras das *frameworks* Dropwizard, Light Rest 4J e Spring Boot.

Frameworks	Prós	Contras
Dropwizard	- Desenvolvido com base na modularidade. - Rápido. - Integração com *frameworks* e bibliotecas de terceiros. - Excelente suporte de monitorização com métricas. - Grande suporte da comunidade.	- Devido à incorporação de *frameworks* e bibliotecas terceiros é preciso ter cuidado com possíveis *bugs*.
Light Rest 4J	- Baixa latência. - Consome pouca memória. - Funciona com outras bibliotecas e *frameworks*. - Desenhado para a escalabilidade. - Organizado em *plugins* para que ocupe o menor espaço possível.	- Má documentação, com algumas secções em falta.
Spring Boot	- Rápido e bastante popular. - Fácil de configurar e manipular. - Modular. - Boa integração com outras bibliotecas. - Grande comunidade. - Boa documentação.	- A fragmentação da documentação, derivado à existência de muitas fontes online, dificulta a procura do conteúdo. - Modificações significativas entre versões.

1. **Pedidos feitos diretamente aos micro-serviços**: neste caso, o cliente faz pedidos diretamente aos micro-serviços necessários, por exemplo através de um *endpoint* público que mapeia no *load balancer* [2] de cada micro-serviço. Os pedidos são então distribuídos pelas várias instâncias desse micro-serviço, caso existam [47].

   Esta abordagem tem algumas desvantagens sendo que as mais evidentes são:

   - Devido à granularidade fina que os micro-serviços apresentam, o cliente pode ter que fazer múltiplos pedidos a diferentes micro-serviços de forma a obter todos os dados pretendidos, o que pode levar a um aumento da latência da operação.
   - Caso exista a necessidade de haver uma modificação num micro-serviço e que, posteriormente, leve à separação em dois micro-serviços distintos, a modificação é bastante difícil dado que o cliente faz uso direto do *endpoint*.

2. **Usar uma API Gateway**: é um servidor que trata de todos os pedidos do cliente, sendo a única forma de acesso ao sistema. A API Gateway oculta a arquitetura interna do sistema e fornece uma API para cada tipo de cliente. Tem ainda a função de agregar os resultados, caso um pedido invoque vários micro-serviços, facilitando a sua interpretação. A Figura 2.4 representa a utilização de uma API Gateway, com dois tipos de clientes distintos, uma aplicação móvel e um cliente desktop.

   Como uma API Gateway encapsula a estrutura interna da aplicação, reduz a comunicação do cliente a um único ponto de acesso. Isto permite diminuir o número de pedidos que o cliente tem de fazer dado que, caso o resultado pretendido envolvesse a chamada a vários micro-serviços, com a Gateway apenas é realizado um. Esta redução também permite simplificar o código do cliente.

[2] Um *load balancer* tem como objetivo encaminhar os pedidos dos clientes para servidores capazes de responder a esses pedidos, de forma a maximizar a velocidade e a capacidade de utilização e de distribuir a carga pelo sistema [30].

Porém existe a possibilidade de se criar um *bottleneck* no desenvolvimento da API Gateway, dado que esta tem de ser atualizada para expor corretamente os *endpoints* dos micro-serviços [36, 47].

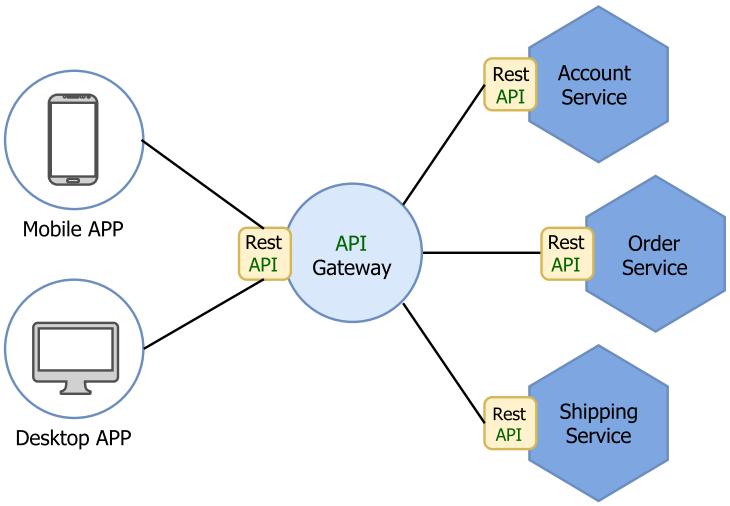

Figura 2.4: Utilização de uma API Gateway com micro-serviços. Adaptado de [47].

Em geral a utilização de uma API Gateway é considerada uma melhor abordagem no desenvolvimento das aplicações, do que a comunicação direta com os micro-serviços [47].

Comunicação - Modelos de Interação (*Communication - Interaction Models*) Os micro-serviços necessitam de comunicar uns com os outros através de um mecanismo de comunicação entre processos (*inter-process communication* - IPC).

Existem diferentes abordagens que se podem utilizar para os micro-serviços interagirem e que podem ser classificadas utilizando dois parâmetros:

- **Número de serviços destino**: existem duas possibilidades, uma mensagem pode ser enviada para apenas um serviço (um-para-um), ou pode ser enviada para vários (um-para-muitos).
- **Tempo da resposta**: a reposta à mensagem enviada é esperada por um tempo determinado (síncrono), ou a resposta pode ser enviada posteriormente e o serviço não fica bloqueado à espera desta (assíncrono).

Estas abordagens de comunicação, para além de serem utilizadas entre os micro-serviços, podem ser utilizadas por um cliente quando pretende interagir com os micro-serviços do sistema.

Tabela 2.2: Estilos de IPC. Adaptado de [47].

		Número de serviços destino	
		Um-para-um	*Um-para-muitos*
Tempo de resposta	*Síncrono*	Pedido/Resposta	-
	Assíncrono	Notificação	Publicador/Subscritor
		Pedido/Resposta assíncrona	Publicador/Respostas assíncronas

Como é possível observar na Tabela 2.2, podem ser feitas combinação entre os diferentes parâmetros identificados anteriormente, sendo que é possível obter vários estilos de IPC. Estes consistem no seguinte:

- **Pedido/Resposta**: o serviço/cliente faz um pedido e fica à espera da resposta.
- **Notificação**: o serviço/cliente faz um pedido, mas não espera uma resposta.
- **Pedido/Resposta assíncrona**: o serviço/cliente faz um pedido e o serviço que a recebe envia a resposta assincronamente.
- **Publicar/Subscrever**: o serviço/cliente publica uma mensagem e esta é consumida pelos serviços interessados.
- **Publicar/Respostas assíncronas**: o serviço/cliente publica uma mensagem e espera uma resposta dos serviços interessados (durante um período de tempo).

Nas diversas tecnologias IPC existentes, os serviços podem utilizar comunicações síncronas, baseadas em pedido/resposta, ou assíncronas, baseadas em mensagens.

A abordagem utilizando mensagens permite que a comunicação seja realizada de forma assíncrona, não fazendo com que o serviço/cliente que comunica fique bloqueado à espera da resposta (caso exista). Permite ainda que as mensagens fiquem armazenadas num *buffer*, sendo que o serviço que as recebe pode processá-las à medida que tem disponibilidade. Tal garante que, eventualmente, todas as mensagens recebidas serão processadas, mesmo que forma mais lenta.

Na abordagem de pedido/resposta (forma síncrona), um serviço/cliente envia um pedido a um serviço e fica à espera de uma resposta. Este pedido pode ser encapsulado utilizando tecnologias que permitam assincronia, mas ao contrário da utilização de mensagens, é esperada uma resposta num dado período de tempo.

Consoante as necessidades do sistema, estes dois tipos de IPC podem ser combinados, tirando proveito dos benefícios de cada um [47].

Sistemas de Suporte (*Supporting Systems*) Numa arquitetura de micro-serviços, para além das tecnologias que permitem o desenvolvimento das funcionalidades principais, é preciso sistemas que possibilitem o armazenamento de dados de forma permanente e a descoberta dos serviços.

Gestão dos Dados (*Data Management*) Numa aplicação monolítica os dados estão todos contidos, de uma forma geral, numa única base de dados relacional. Isto permite que o sistema contenha as propriedades ACID e, caso seja preciso atualizar múltiplas tabelas como uma operação única, isto é possível através do uso de transações.

Geralmente, cada micro-serviço possui a sua própria base de dados para permitir que os micro-serviços sejam independentes uns dos outros e facilitar o desenvolvimento, o *deployment* e a evolução de cada um deles de forma independente. Podem ser utilizados diferentes tipos de base de dados (SQL ou NoSQL), dependendo dos objetivos.

Apesar dos benefícios, há um nível de complexidade maior quando é necessário consultar ou atualizar dados que estão dispersos nas diferentes bases de dados, pois não é possível fazer *queries* a base de dados externas ao próprio micro-serviço, tendo tal que ser realizado com recurso a APIs.

Problemas neste tipo de arquitetura incluem a implementação de transações com consistência dos dados dos vários micro-serviços, e fazer *queries* que devolvam dados de múltiplos micro-serviços e que usam tipos distintos de bases de dados (SQL e NoSQL).

Uma das soluções possíveis é ter uma arquitetura orientada a eventos, em que o micro-serviço publica um evento quando há alguma alteração (inserção, atualização ou eliminação) numa das tabelas da sua própria base de dados. Os outros micro-serviços interessados nesses dados fazem uma subscrição a esses eventos e, quando os recebem, executam uma dada lógica, podendo publicar outros eventos. Esta troca de eventos entre os micro-serviços é realizada através de um componente denominado *Message Broker*.

A arquitetura de micro-serviços orientada a eventos permite que a aplicação veja estados inconsistentes e que requer a implementação de código que lide com esse problema. Por exemplo, um micro-serviço atualizou um dado valor na sua base de dados mas o evento que foi publicado ainda não foi processado pelos outros micro-serviços que fizeram a sua subscrição, tendo por isso valores incoerentes nas suas bases de dados.

Tal acontece porque as operações de escrever na base de dados e, publicar o evento respetivo, são realizadas independentemente. Não é, portanto, certo que, a seguir à escrita na base de dados, a operação de publicar o evento se concretize. Caso esta falhe, os micro-serviços que subscreveram este evento não o recebem, causando a inconsistência dos dados.

Este problema de consistência é derivado da falta de transações tradicionais, tendo que ser implementados mecanismos que garantam que as duas operações de alteração e subsequente notificação sejam vistas como uma operação atómica, ou seja, sejam executadas como uma só do ponto de vista externo do sistema [47].

Este problema pode ser ultrapassado através das seguintes abordagens:

- **Publicar eventos com recurso a transações locais**: é necessário ter uma tabela auxiliar, em cada micro-serviço, onde vão sendo escritos os eventos quando se efetua alguma atualização numa dada tabela. Estas duas operações são feitas dentro de uma transação, garantido que ambas são escritas, ou nenhuma delas é. Existe também um processo que vai fazendo *queries* à tabela de eventos e os publica no *Message Broker*. Esta abordagem tem limitações, não podendo ser utilizada em base de dados NoSQL, dado que estas (tipicamente) não têm o conceito de transação [42, 47].

- **Utilizar *event sourcing***: ao invés de escrever na base de dados o estado atual de uma tabela, os eventos das alterações à tabela são guardados num componente denominado *Event Store*, que é uma base de dados de eventos. O *Event Store* tem um comportamento similar ao *Message Broker*, que possibilita aos micro-serviços subscreverem eventos. A Figura 2.5 mostra como seria a utilização de *event sourcing* ao invés da utilização de uma tabela normal.

  Apesar de resolver o problema da consistência dos dados, o *event sourcing* tem desvantagens para fazer consultas aos dados, tendo por isso que ser usado *command query responsability separation* (CQRS) [38, 47]. O CQRS é uma arquitetura em que são utilizadas diferentes base de dados para operações de leitura e escrita.

  Após ser inserido algum valor (evento) no *Event Store*, a base de dados de leitura pode ser atualizada de forma síncrona ou assíncrona. Se for realizada de forma síncrona é possível oferecer consistência forte ou consistência eventual. Se for realizada de forma assíncrona será consistência eventual [59].

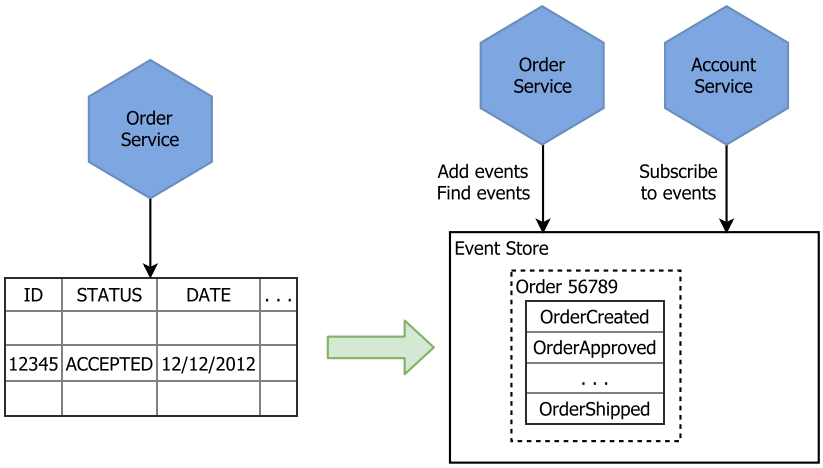

Figura 2.5: Utilização de *event sourcing*. Adaptado de [47].

Descoberta de serviços (*Service Discovery*) Num sistema com a arquitetura de micro-serviços, como foi referido anteriormente (secção 2.2.3.2) muitas das vezes existe a necessidade de comunicação entre vários micro-serviços. Ao contrário de uma aplicação monolítica, onde os recursos têm uma localização estática (ou muda muito raramente ao longo do tempo), numa arquitetura deste género, a localização dos recursos é atribuída dinamicamente devido, por exemplo, a falhas ou a atualizações [47].

Para descobrir serviços existem dois padrões principais: descoberta no lado do cliente, ou descoberta no lado do servidor.

Ambos os padrões para localizarem serviços necessitam de um componente denominado *service registry*. Este é um componente que tem uma base de dados, onde são guardadas as localizações dos micro-serviços, que depois é contactado pelos clientes ou outros micro-serviços quando estes necessitarem. Existem duas formas principais em que os serviços são registados e eliminados do *service registry*: os micro-serviços podem-se registar eles mesmos, por exemplo, através de uma API REST e atualizar a sua localização, bem como elimina-la; ou pode existir outro componente que analisa os micro-serviços e os vai registando no *service registry* [46, 47].

- **Descoberta do lado do cliente**: o cliente contacta um *service registry* em que devolve a localização dos recursos (micro-serviços). O cliente faz então o pedido ao micro-serviço que pretende e, caso haja vários serviços do mesmo tipo, este escolhe um aleatoriamente para tentar distribuir a carga entre as várias instâncias do mesmo tipo de serviço (faz o papel de *load balancer*).

  Este padrão tem o inconveniente de fazer com que o cliente fique dependente do *service registry*, tendo que ser implementada uma lógica diferente para clientes diferentes (usando diferentes linguagens de programação e *frameworks*) [37, 47].

- **Descoberta do lado do servidor**: o cliente ou o serviço contactam o *load balancer*, em que este faz uma pesquisa no *service registry* para obter a localização do recurso, e em seguida redireciona o pedido para o serviço correspondente. Este padrão tem o beneficio de abstrair o cliente do *service registry*, dado que o cliente apenas contacta o *load balancer*. Apesar disso, tem o senão de ter que se implementar e gerir um *load balancer*, um componente que tem de estar sempre disponível [43, 47].

2.2.3.3 *Deployment*

Plataforma (*Platform*) A arquitetura de micro-serviços é uma arquitetura *cloud-native* [3], sendo por isso a *cloud* a plataforma principal para o seu *deployment* [22]. Esta é caracterizada consoante onde se localizada a infraestrutura, podendo ser do tipo ***public cloud***, ***private cloud*** ou ***hybrid cloud***, descritas na secção 2.3.1.

Gestão (*Management*) A possibilidade da existência de inúmeros micro-serviços numa aplicação com esta arquitetura, leva que a gestão das instâncias dos micro-serviços tenha que ser realizada com a menor intervenção humana possível, permitindo um comportamento correto, por exemplo, quando ocorrem falhas ou existe a necessidade de aumentar recursos de um dado micro-serviço.

Esta gestão pode ser conseguida através de soluções já fornecidas pela própria *cloud*, ferramentas externas ou serem desenvolvidos mecanismos próprios para uma aplicação.

Por exemplo, os principais fornecedores de serviços na *cloud*, como a Amazon, Microsoft e Google disponibilizam funcionalidades como o *auto scaling*, permitindo que haja um aumento ou diminuição de recursos automaticamente em determinados serviços, consoante o necessário.

Em relação a ferramentas externas aos fornecedores de serviços existe, por exemplo, a RightScale [3] que disponibiliza funcionalidades para a gestão dos recursos na *cloud* [16].

2.2.3.4 Tempo de execução (*Runtime*)

Virtualização (*Virtualization*) Uma aplicação de micro-serviços contém vários serviços, em que cada um deles pode ser desenvolvido utilizando linguagens de programação e *frameworks* distintas. Cada micro-serviço tem, portanto, requisitos específicos de *deployment*, monitorização e recursos (ex.: CPU e RAM).

Existem diversos padrões possíveis em relação ao *deployment* de micro-serviços:

- **Uma instância de serviço por *Host***: Uma única instância de um serviço encontra-se isolada no seu próprio *Host*. Sendo que existem dois tipos específicos para este tipo de padrão:

  - **Instância de serviço por máquina virtual**: cada serviço é instalado numa VM, sendo que depois é feito o *deployment* dessa VM.

    Um dos benefícios desta abordagem é o isolamento completo de cada serviço em relação aos outros serviços, possibilitando a definição dos recursos (CPU, RAM) que cada VM pode utilizar. Permite também, ao utilizar uma infraestrutura na *cloud*, a utilização de estratégias disponíveis de *load balancing* (balanceamento da carga) e *auto scaling* (escalonamento automático) das máquinas virtuais.

    Apesar destas vantagens, este padrão tem o inconveniente de ser menos eficiente em termos de recursos quando comparado com o padrão de múltiplas instâncias (referido no próximo ponto), pois tem o custo adicional de criar a VM, o que inclui o sistema operativo. Fazer o *deployment* de uma nova versão de um serviço também é um pouco lento, pois tem que se instalar o serviço na VM [45, 47].

[3] RightScale: https://www.rightscale.com/

  - **Instância de serviço por *container***: Cada serviço corre no seu próprio *container* [4]. Os serviços são instalados como uma imagem de um *container*, depois podem ser lançados vários *containers* numa máquina (física ou virtual). Este padrão tem vantagens similares ao padrão de uma instância por máquina virtual, mas é uma tecnologia mais leve e rápida, sendo bastante rápido a iniciar um serviço. Contudo existem desvantagens, na utilização deste padrão. Ao contrário do padrão de uma instância por VM, no padrão de *containers* existe a partilha do núcleo do sistema operativo entre os *containers*, o que não é tão seguro [44, 47].

- **Múltiplas instâncias de serviços por *Host***: Consiste em ter várias instâncias de diferentes serviços numa única máquina (física ou virtual).

  Este padrão de *deployment* tem diversas vantagens, sendo que a principal é a eficiência dos recursos utilizados, quando comparado com o padrão de 'Uma instância por *Host*'. Um outro benefício é a sua facilidade de realizar o *deployment* de uma instância de um serviço, sendo que para isso apenas é necessário copiar o executável ou o código fonte para o *Host*, e inicializar a aplicação.

  Porém também existem desvantagens ao utilizar este padrão. Ao ter instâncias de serviços diferentes na mesma máquina, estas não se encontram totalmente isoladas umas das outras, podendo haver conflitos nos recursos e software que cada uma utiliza. É também difícil limitar os recursos que cada serviço pode utilizar e, consequentemente monitorizar os recursos utilizados por cada um deles [41, 47].

Controlo (*Control Loops*) Num sistema com uma arquitetura de micro-serviços, têm de existir mecanismos que permitam controlar os serviços que se encontram em execução, através da sua monitorização e da análise dos dados recolhidos pela monitorização, por exemplo.

O exemplo seguinte, que consiste na utilização de um micro-serviço para a deteção de anomalias e para a análise das suas causas em aplicações móveis e IoT, descrito em [2] é uma das possíveis abordagens.

As operações efetuadas por dispositivos móveis e IoT podem produzir grandes quantidades de dados, em que é necessária a análise de alguns desses dados para identificar e corrigir problemas. A capacidade de deteção de anomalias pode identificar anomalias operacionais como uma alta latência da aplicação, ou medições erradas em sensores.

A capacidade de análise pode ajudar a identificar a causa da anomalia. Estas funcionalidades podem ser usadas para perceber certos comportamentos da aplicação, como por exemplo:

- **Medição do desempenho**: o micro-serviço pode ser útil para detetar rapidamente se uma aplicação está lenta devido ao aumento do tempo de resposta de um serviço de *backend*. A análise da causa do problema pode permitir que depois sejam aplicadas ações que corrijam o mesmo.

- **Análise de falhas e de mal funcionamento**: um exemplo da análise da causa do problema, pode ser a perceção que uma aplicação móvel bloqueia na presença de problemas na rede, por exemplo, enquanto o utilizador se move entre duas torres celulares adjacentes.

[4] Um *container* é um mecanismo de virtualização ao nível do sistema operativo, que permite a existência de múltiplas instâncias isoladas. Um *container* permite isolar o código, contendo bibliotecas e as definições num único pacote, minimizando conflitos entre diferentes softwares que se encontram na mesma infraestrutura [9].

Um exemplo de mal funcionamento pode ser o aumento do tempo de resposta de uma aplicação móvel quando uma nova versão do sistema operativo é instalada.

2.2.4 Transição de uma aplicação monolítica para micro-serviços

A transição de uma aplicação monolítica para micro-serviços é um processo que deve ser ponderado para verificar se será benéfico para o sistema utilizar este tipo de arquitetura. Como em qualquer refatorização, esta deve ser feita de forma incremental, de maneira a que os riscos sejam mínimos, não aplicando todos os esforços em modificar todos os componentes da aplicação, pois não é a abordagem correta e tem grandes possibilidades de falhar.

A utilização de metodologias incrementais consiste, por exemplo, em ir adicionando novas funcionalidades e em criar extensões de funcionalidades já existentes em forma de micro-serviços, modificando a aplicação monolítica de uma forma complementar.

Ao logo do tempo, as funcionalidades implementadas pela aplicação monolítica serão assim mínimas até desaparecem ou poderem ser migradas para um novo micro-serviço.

Existem várias estratégias para refatorizar uma aplicação em micro-serviços [47]:

1. **Novas funcionalidades através de micro-serviços**: as novas funcionalidades a implementar devem ser feitas num único novo micro-serviço, de forma a não tornar a aplicação monolítica mais complexa.

   Com esta estratégia é necessário adicionar dois componentes:

   a) um *request router*, similar à API Gateway (descrita em 2.2.3.2), que tem como objetivo mapear os pedidos, sendo que as novas funcionalidades serão redirecionadas para o micro-serviço e as antigas para a aplicação monolítica;

   b) *glue code* que tem como objetivo fazer a integração entre o micro-serviço e a aplicação monolítica, nomeadamente ao nível dos dados pertencentes à aplicação monolítica, quer seja para fazer leituras ou escritas.

      Existem 3 formas do micro-serviço aceder aos dados da aplicação monolítica:

      - Invocar uma API fornecida pela aplicação monolítica;
      - Aceder à base de dados da aplicação monolítica diretamente;
      - Manter uma cópia dos dados, que é sincronizada com os dados da aplicação monolítica.

   Esta estratégia apenas é útil para que a aplicação monolítica não aumente de dimensão e complexidade. Para resolver problemas que a aplicação monolítica possa apresentar é necessário utilizar as outras estratégias.

2. **Separar o *frontend* do *backend***: consiste em separar a camada de apresentação, da lógica de negócio e da camada de acesso aos dados, dando origem a duas aplicações. A primeira tem apenas a camada de apresentação que, por sua vez, comunica com uma segunda aplicação que contém a lógica de negócio e a camada de acesso aos dados com recurso a APIs. Esta separação apenas permite simplificar o desenvolvimento da camada de *frontend* e de *backend*, originando duas aplicações monolíticas que podem ser de difícil gestão, tendo para isso que ser aplicada a estratégia seguinte.

3. **Extrair os serviços**: esta estratégia consiste na transformação de módulos da aplicação monolítica a micro-serviços. Ao ir passando os módulos em micro-serviços, o tamanho da aplicação monolítica diminui, possibilitando que no fim esta passe a um outro micro-serviço, ou mesmo desapareça.

    - **Prioritizar os módulos a serem convertidos em serviços**: uma boa abordagem é começar pelos módulos que sejam mais fáceis de extrair. Isto permite ganhar experiência no processo de conversão de módulos em serviços.

      Em seguida devem ser extraídos os módulos que mudam frequentemente, para permitir que o desenvolvimento e o *deployment* seja feito de forma independente, acelerando o desenvolvimento.

    - **Extrair um módulo**: o primeiro passo para extrair um módulo, é definir um interface *coarse-grained* entre a o módulo e a aplicação monolítica. A API deve ser bidirecional dado que o módulo necessitará de dados da monolítica e o contrário também. Quando esta interface estiver implementada, o módulo torna-se um serviço autónomo. Depois é necessário implementar uma API que permita a comunicação através de um mecanismo de comunicação entre processos (IPC), fazendo com que o serviço se torne independente.

2.2.5 Utilização da arquitetura de micro-serviços na prática

Recentemente, diversas empresas realizaram a transição da arquitetura das suas aplicações e serviços para a utilização de micro-serviços como modelo arquitetural. Em baixo são apresentadas algumas empresas que fizeram esta transição e quais os benefícios que encontram com a nova arquitetura.

2.2.5.1 Yelp

A Yelp [5] é o caso duma empresa que fez a transição de uma aplicação monolítica para uma aplicação com uma arquitetura baseada em serviços. Esta mudança foi necessária devido ao facto do código da aplicação monolítica, 'yelp-main', ser demasiado extenso (vários milhões de linhas) e da equipa ter aumentado, fazendo que se demorasse muito tempo a efetuar modificações ou a adicionar novo código.

Foi referido em [21] que a aplicação monolítica, 'yelp-main' ainda possui bastante código, sendo difícil que a mesma desapareça, mas tem passado a ser mais uma aplicação de *frontend*, responsável pela agregação e visualização de dados dos serviços de *backend*.

Interfaces Os serviços expõem interfaces HTTP em que retornam dados estruturados JSON. Um dos grandes benefícios de utilizar HTTP e JSON é que existem ferramentas para *debug*, *cache* e balanceamento de carga. No entanto, não existe uma solução standard para definir as interfaces dos serviços, fazendo com que seja difícil a sua especificação e verificação.

Este problema foi resolvido através do Swagger que fornece uma linguagem para documentar a interface dos serviços HTTP/JSON. Foi utilizado o swagger-py para criar automaticamente um cliente em Python para o mesmo serviço através de uma especificação Swagger. Foi também

[5] Yelp: https://www.yelp.com/

utilizado o Swagger UI para fornecer uma diretoria centralizada para as interfaces dos serviços, permitindo que seja facilmente descoberto que serviços estão disponíveis.

Stack dos serviços A maior parte dos serviços foram desenvolvidos em Python utilizando diferentes componentes *open-source*. É utilizada a Pyramid como *web framework* e o SQLAlchemy para a camada de acesso à base de dados, tendo como base uWSGI.

Um serviço pode ser escrito em qualquer linguagem, permitindo que seja escolhida a que melhor corresponder às necessidades. Por exemplo, a equipa de pesquisa desenvolveu múltiplos serviços em Java, ao invés da utilização de Python.

Repositório de dados Grande parte dos serviços necessita de armazenar dados, tendo havido a flexibilidade de escolher o tipo de base de dados que melhor se enquadrava num dado serviço. Foram utilizados o MySQL, o Cassandra e o ElasticSearch. A implementação dos repositórios de dados é privada ao próprio serviço, permitindo que a longo prazo esta possa ser alterada ao nível da representação dos dados, ou mesmo a tecnologia.

Muitos dos dados principais, como de negócio, de utilizadores e das *reviews* encontram-se na base de dados da 'yelp-main'. A extração deste tipo de dados relacionais para serviços sendo bastante complexa, optou-se pela utilização de uma API interna para os serviços acederem a estes dados.

Descoberta de serviços Um dos problemas principais deste tipo de arquiteturas é a descoberta da localização das instâncias de um dado serviço. Primeiramente foram configurados manualmente *load balancers* HAProxy em cada um dos *data centers*, e foi colocado o endereço IP virtual em ficheiros de configuração. Verificou-se que esta abordagem não escalava, pois era bastante trabalhosa e propensa a erros, tendo sido detetados problemas ao nível da performance, devido aos *load balancers* estarem a ficar sobrecarregados.

Estes problemas foram resolvidos através da mudança para um sistema de descoberta de serviços baseado no SmartStack. Cada cliente tem uma instância HAProxy que está ligada ao localhost, que é configurado dinamicamente a partir das informações armazenadas no ZooKeeper [6]. Um cliente pode depois utilizar um serviço conectando-se ao *load balancer* do localhost, que redireciona o pedido para uma instância do serviço. Esta abordagem provou que é altamente fiável e a maior parte dos serviços de produção está a utilizá-la [21].

2.2.5.2 HubSpot

A HubSpot [7] desenvolve uma plataforma de marketing e vendas para fornecer uma maior presença online dos seus clientes.

Cada equipa é composta por um líder técnico e dois *developers* que trabalham com um gestor de produto e um designer. As equipas são pequenas para evitar problemas de escalamento e demasiadas comunicações, permitindo também que o líder técnico esteja focado no produto e possa auxiliar os *developers*.

Os produtos são compostos por mais de 300 serviços, e dezenas de aplicações de *frontend*. A maior parte dos micro-serviços são desenvolvidos em Java com a *framework* Dropwizard, e a parte

[6] ZooKeeper: é um serviço centralizado para manter informações de configuração, nomes, fornecer sincronização distribuída e serviços de grupo [53].

[7] HubSpot: https://www.hubspot.com/

de *frontend* utiliza Backbone e React em CoffeScript. Os serviços comunicam através de APIs REST ou de um sistema de mensagens como o Apache Kafka.

Esta arquitetura permite rápidas iterações, dado que existem mais de 1000 unidades que podem ser *deployed* separadamente, permitindo que escalem independentemente.

Existe uma complexidade adicional para perceber qual o desempenho de uma arquitetura distribuída, bem como em gerir determinadas configurações (ex.: serviços a utilizarem versões diferentes de uma biblioteca partilhada). Esta mudança de uma aplicação monolítica, para equipas pequenas, pode não ser benéfica até que seja atingindo um certo tamanho no código ou no número de *developers*.

Uns dos principais requisitos para conseguir *deploys* pequenos e seguros são as *feature flags* (sinalizadores de funcionalidades). Uma funcionalidade desenvolvida ao longo do tempo pode ser integrada de forma segura no código principal tendo um sinalizador. A principal vantagem é que assim pode ser controlada quem tem acesso à nova funcionalidade, assegurando que em determinado ponto estará pronta para produção. Uma funcionalidade pode ser escondida em qualquer instante, sem que para isso seja necessário outro *deploy*.

Para que os *deploys* sejam seguros é necessário obter informações do estado dos serviços. O projeto Rodan, permite que sejam recolhidas métricas standard (ex.: pedidos/segundo), bem como outras definidas pelos *developers*.

O EC2 da Amazon tem sido bastante utilizado pela HubSpot. Com a utilização do Apache Mesos e de uma aplicação que foi desenvolvida pela HubSpot, denominada Singularity, que permite gerir os *clusters* EC2 de uma forma mais otimizada, permitindo uma maior densidade de serviços por servidor, reduzindo custos. Também permite que os *developers* não tenham intervenção nas instâncias, sendo a plataforma a lidar com determinados problemas [4].

2.3 *Cloud Computing*

A computação em *Cloud* (*Cloud Computing*) tem tido um grande impacto nos últimos anos, pela forma como é realizada a computação e o armazenamento de dados (e não só), possibilitando a disponibilização de recursos (CPU, armazenamento) em grande escala e, a pedido. Tal estava anteriormente restrito a empresas, universidades ou institutos governamentais que possuíssem *data centers*, e não ao público em geral.

A computação em *Cloud* foi precedida, na década de 1990, pela computação em Grid (*Grid Computing*) que permitia aos utilizadores obter computação a pedido para resolver problemas científicos complexos, por acesso a recursos computacionais na maioria de Universidades e instituições governamentais [12]. A computação em *Cloud* diferiu desse modelo ao utilizar tecnologias de virtualização a nível aplicacional e hardware, para fornecer e partilhar recursos de uma forma dinâmica [60], possibilitando um modelo de negócio vantajoso para grandes empresas como a Amazon, Google, Microsoft, etc.

De acordo com *The National Institute of Standards and Technology* (NIST), *Cloud Computing* é um modelo de computação que permite, de uma forma prática, o acesso a um conjunto de recursos de computação partilhados (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem rapidamente ser configurados e fornecidos com o mínimo esforço de gestão ou interação do fornecedor de serviços [25].

O modelo de computação em *Cloud* fornece algumas funcionalidades distintas/únicas [25]:

- ***Self-service* e a pedido**: um utilizador pode obter capacidades computacionais consoante as suas necessidades, de forma automática, sem a necessidade de intervenção humana por parte do fornecedor de serviço.
- **Acessível pela rede**: os serviços estão disponíveis na rede e são acessíveis através de mecanismos standards com o recurso a smartphones, tablets ou computadores.
- **Agrupamento dos recursos**: os recursos computacionais do fornecedor são agrupados num modelo *multi-tenant* com diferentes recursos físicos e virtuais, que são atribuídos dinamicamente consoante os pedidos dos utilizadores. Na maior parte das vezes, o utilizador não tem controlo sobre a localização exata dos recursos, podendo em alguns casos especificar a localização num nível de abstração mais alto (por exemplo, ao nível do país).
- **Elasticidade rápida**: as capacidades podem ser requisitadas e libertadas elasticamente (por vezes de forma automática), para se escalar rapidamente de acordo com a necessidade. Para o utilizador, as capacidades disponíveis dos recursos parecem ser ilimitadas e podem ser definidas em qualquer momento.
- **Serviço monitorizado**: o uso dos recursos (armazenamento, processamento, largura de banda, entre outros) pode ser monitorizado, possibilitando que os sistemas de *cloud* controlem e otimizem automaticamente a sua utilização.

O modelo de negócio que a computação em *Cloud* utiliza é um modelo orientado ao serviço, ou seja, os recursos, quer a nível de hardware e/ou software, são disponibilizados como serviços que o utilizador requisita quando necessitar [60]. Os serviços podem ser agrupados em 3 categorias, consoante o tipo de serviço que é fornecido ao utilizador [25, 60]:

- ***Infrastructure as a Service* (IaaS)**: são fornecidos ao utilizador recursos de infraestruturas, como processamento, armazenamento, redes, normalmente através de máquinas virtuais (VMs), onde pode ser instalado e executado qualquer tipo de software, desde sistemas operativos a aplicações.
- ***Platform as a Service* (PaaS)**: são fornecidas capacidades para o utilizador realizar o desenvolvimento e *deployment* de aplicações utilizando linguagens de programação, *frameworks* e bibliotecas suportadas pelo fornecedor do serviço.
- ***Software as a Service* (SaaS)**: é fornecido ao utilizador a capacidade de executar aplicações através da Internet, em que estas podem ser acedidas através dum browser ou da instalação de uma aplicação cliente.

2.3.1 Modelo de *Deployment*

Consoante o local onde se encontram as infraestruturas, é possível caracterizar o tipo de *cloud*.

Os principais modelos de *deployment* consistem em [25, 60]:

- ***Public cloud***: a infraestrutura é disponibilizada de forma aberta, fornecendo recursos como serviços, ao público em geral. Pode ser gerida por qualquer tipo de organização, mas apenas está fisicamente nas instalações do fornecedor da *cloud*.

- ***Private cloud***: a infraestrutura é apenas para uso exclusivo de uma única organização. Esta pode ser gerida pela própria organização ou por uma entidade externa. Sendo para uso exclusivo, permite obter um maior controlo sobre a performance, fiabilidade e segurança.

- ***Hybrid cloud***: a infraestrutura é uma combinação das infraestruturas de *cloud* pública e privada, de forma a mitigar as limitações que cada uma apresenta. Parte dos serviços são executados na *cloud* privada, e os restantes na *cloud* pública. Fornecem um melhor controlo sobre os dados que *clouds* públicas, e ao mesmo tempo permitem a elasticidade dos recursos.

2.3.2 Virtualização de recursos

A gestão de recursos em *data centers* é uma tarefa bastante complexa devido à quantidade de recursos disponíveis e cresce com o aumento do número de utilizadores e de serviços fornecidos. É ainda afetada pela questão da heterogeneidade do hardware e do software utilizado, assim como pela existência de possíveis falhas que possam ocorrer nos componentes.

A solução para realizar esta gestão passa pela virtualização dos recursos, uma técnica utilizada na computação em *Cloud* que permite que as tarefas de gestão de alguns recursos sejam simplificadas. Por exemplo, com a utilização de máquinas virtuais (VMs), o estado de uma VM pode ser guardado e migrada para outro servidor de forma a balancear a carga.

A partilha de recursos num ambiente de máquina virtual requer o suporte do hardware e processadores compatíveis. Os recursos como CPUs, memórias, armazenamento, I/O e largura de banda são partilhados entre as máquinas virtuais.

Existem duas técnicas de virtualização: a virtualização completa e a para-virtualização.

- A **virtualização completa** consiste numa máquina virtual em que esta é executada numa cópia exata do hardware. Tal só é possível quando a abstração do hardware fornecida pela VMM [8] é uma réplica exata do hardware físico. Neste caso, qualquer sistema operativo é executado no hardware sem modificações, desde que possua uma arquitetura que seja virtualizável. As VMMs VMware são exemplos de virtualização completa.

- A **para-virtualização** consiste numa máquina virtual que possui uma cópia modificada do hardware. São necessárias algumas modificações nos sistemas operativos *guest*, porque a abstração para o hardware fornecida pela VMM não suporta todas as funções do hardware físico. A Xen e a Denali são exemplos de VMMs baseadas em para-virtualização.

  Existem razões para que seja adotada a para-virtualização:

  - Alguns aspetos do hardware não são virtualizáveis;
  - Melhorar a performance;
  - Fornecer uma interface mais simples.

  Este tipo de virtualização é também considerado, pois existem algumas arquiteturas, como x86, que não são facilmente virtualizáveis [23].

[8] Uma *Virtual Machine Monitor* (VMM) é um software que permite a criação e a gestão de máquinas virtuais (VMs), sendo também conhecida por *Virtual Machine Manager* e *Hypervisor* [52].

2.3.3 Gestão de recursos e agendamento

A gestão de recursos é uma funcionalidade essencial em qualquer sistema, e está relacionada com três critérios base de avaliação de um sistema: performance, funcionalidade e custo. Uma gestão de recursos ineficiente impacta negativamente e de forma direta a performance e o custo do sistema, e indiretamente a sua funcionalidade [24].

2.3.3.1 Gestão de recursos

Uma *cloud* é um sistema com um número bastante considerável de recursos partilhados que são sujeitos a pedidos inesperados. Assim a gestão destes recursos requer políticas complexas e decisões para a sua otimização [24].

As estratégias para a gestão de recursos diferem consoante o tipo de serviço fornecido: IaaS, PaaS e SaaS. Em todos os casos, os fornecedores de serviços são afetados com flutuações de carga, um comportamento que desafia a elasticidade da *cloud*. Em certos casos, quando um pico de carga pode ser previsto, os recursos podem ser provisionados à priori. Em casos em que não são esperados picos de carga, a situação é mais complicada, podendo ser utilizado o *auto scaling* se: 1. Existir um conjunto de recursos que podem ser libertados e alocados a pedido; 2. Existir um sistema de monitorização que permita decidir em tempo real a realocação de recursos. O *auto scaling* é suportado em serviços PaaS, mas é mais complicado de implementar em IaaS devido à falta de standards.

Existe também a necessidade de políticas autonómicas devido à dimensão do sistema, o grande número de pedidos aos serviços, a grande quantidade de utilizadores e à imprevisibilidade de carga, de forma a diminuir a intervenção humana na gestão do sistema.

As políticas de gestão de recursos na *cloud* podem ser divididas em cinco classes: 1. Controlo de admissão; 2. Alocação de capacidade; 3. Balanceamento de carga; 4. Otimização de energia; 5. Qualidade de Serviço (*Quality of Service* - QoS).

O objetivo da política de controlo de admissão é prevenir que o sistema aceite uma carga de trabalho que pode prejudicar outro trabalho em execução ou que tenha sido, entretanto, requisitado. Limitar a carga de trabalho requer o conhecimento do estado global do sistema, o que num sistema dinâmico é difícil porque, quando esse conhecimento está disponível, no melhor dos casos, pode já estar desatualizado.

A alocação de capacidade consiste em alocar recursos para instâncias individuais. O balanceamento de carga e a otimização de energia estão ambos ligados, e afetam o custo do fornecimento dos serviços.

Um dos objetivos na computação em *Cloud* é manter o custo do fornecimento de um dado serviço no valor mais baixo possível, e uma das formas de o realizar é minimizar o consumo de energia. Assim sendo, o 'balanceamento de carga' pode ter um significado que não traduz uma utilização de recursos adequada em termos de custos. Em vez de se distribuir a carga uniformemente por todos os servidores, pode ser preferível utilizar um número mais reduzido de servidores e distribuir o trabalho por esses, deixando os restantes em *standby*, levando a que seja consumida menos energia.

2.3.3.2 Agendamento

O agendamento é uma funcionalidade crítica na gestão de recursos na *cloud*. Este é responsável pela partilha e multiplexagem a vários níveis. Um servidor pode ser partilhado entre várias máquinas virtuais, em que cada VM pode suportar várias aplicações, e cada aplicação pode consistir em múltiplos *threads*. O agendamento de CPU suporta a virtualização de um processador, em que cada *thread* atua como um processador virtual; um link de comunicação pode ser multiplexado entre vários canais virtuais, um para cada fluxo [24].

Um algoritmo de agendamento, para além de cumprir os seus objetivos, deve ser eficiente, justo e *starvation-free*. Por exemplo, os objetivos de um *scheduler* para um sistema de *batch* são maximizar a taxa de transferência (o número de trabalhos completos por unidade de tempo) e minimizar o tempo de resposta (o tempo entre a submissão de um trabalho e a sua conclusão).

Alguns *schedulers* são preemptivos, permitindo que uma tarefa de alta prioridade interrompa a execução de outra com baixa prioridade; e outros são não-preemptivos.

2.3.4 Infraestruturas de *cloud*

Existem bastantes ofertas no que diz respeito a infraestruturas de *cloud* públicas, sendo que as mais utilizadas são as da Amazon, Microsoft e Google, sendo esperado que tenham 76% da receita neste mercado em 2018 [5].

A descrição que se segue ilustra a diversidade de recursos disponibilizados pelos fornecedores de *cloud* principais (a Amazon com a Amazon Web Services [9] (AWS), a Microsoft com o Azure [10] e a Google com a Google Cloud Platform [11] (GCP)) e que permite que diferentes micro-serviços possam ser desenvolvidos, e depois executados, usando tecnologias diversas e que sejam as mais adequadas às funcionalidades pretendidas.

2.3.4.1 Computação

A Amazon disponibiliza o Elastic Compute Cloud (EC2) que é a base de todos os serviços de computação. Funciona através de máquinas virtuais que podem ser configuradas pelo utilizador ou através de definições pré-configuradas. O EC2 fornece bastantes opções, com suporte para Windows e Linux, instâncias *bare metal* (em pré-visualização), instâncias GPU, computação de alta-performance, *auto scaling* (escalamento automático), entre outros.

A Amazon ainda disponibiliza vários serviços de *containers*, como o Docker, Kubernetes, e o seu próprio serviço Fargate que automatiza a gestão de servidores e *clusters*.

A Microsoft disponibiliza os serviços Virtual Machines e Virtual Machine Scale Sets. O serviço de computação Virtual Machines suporta Linux, Windows Server, SQL Server, Oracle, IBM e SAP, capacidades híbridas de cloud e suporte integrado para software Microsoft. Este serviço permite opções para computação de alta-performance e GPU, como também instâncias otimizadas para inteligência artificial e *machine learning*.

O Azure permite *auto scaling* através de Virtual Machine Scale Sets. Tem ainda dois serviços de *containers*, o Azure Container Service que é baseado no Kubernetes e o Container Services que utiliza o Docker Hub e Azure Container Registry para a gestão. Possui também um serviço

[9] AWS: https://aws.amazon.com
[10] Azure: https://azure.microsoft.com
[11] GCP: https://cloud.google.com

único denominado Service Fabric, concebido para aplicações com uma arquitetura baseada em micro-serviços.

A Google disponibiliza o Google Compute Engine (GCE) que é idêntico ao serviço EC2 da AWS. Em relação a *containers* é disponibilizado o Kubernetes Engine, devido ao facto da participação da Google no projeto do Kubernetes [19, 58].

2.3.4.2 Armazenamento e base de dados

Para o armazenamento, a Amazon oferece o Simple Storage Service (S3) para o armazenamento de objetos, o Elastic Block Storage (EBS) para armazenamento persistente, utilizado com o EC2, e o Elastic File System (EFS) para o armazenamento de ficheiros.

Em relação às bases de dados, a Amazon tem uma base de dados SQL própria, denominada Aurora, o Relational Database Service (RDS) que permite a utilização de várias base de dados relacionais, a base de dados NoSQL DynamoDB, base de dados em memória ElastiCache, entre outros. A Amazon não oferece um serviço de backup, mas fornece o Glacier que permite arquivar os dados. A Storage Gateway pode ser utilizada para se configurar os processos backups.

A Microsoft tem como serviços básicos de armazenamento o Blob Storage para armazenamento de objetos de dados não estruturados, Queue Storage para grandes volumes de cargas de trabalho, File Storage e Disk Storage. Tem também a Data Lake Store que é útil para aplicações *big data*. Existem também várias opções de base de dados no Azure. Têm três opções baseadas em SQL: SQL Database, Database for MySQL e Database for PostgreeSQL.

A Microsoft tem um serviço de Data Warehouse, assim como Cosmos DB e Table Storage para NoSQL. A Redis Cache é um serviço de base de dados em memória e o Server Stretch Database é um serviço de armazenamento híbrido desenhado especificamente para as aplicações que utilizam o SQL Server. Existe também um serviço de backup.

A Google tem um conjunto menor de serviços de armazenamento. Tem o Cloud Storage que é um serviço para o armazenamento de objetos e também tem a opção de Persistent Disk. Em relação às bases de dados, a GCP tem a Cloud SQL e uma base de dados relacional Cloud Spanner. Para base de dados NoSQL tem a Cloud Bigtable e Cloud Datastore, mas não fornece serviços de backup [19, 58].

2.3.4.3 Localização

Em relação aos locais onde estão presentes, ambos têm infraestruturas que cobrem o mundo [58]. A Amazon é quem fornece maior cobertura com 49 zonas de disponibilidade, tendo planos para disponibilizar mais 12 zonas [1]. A Microsoft e a Google têm ambas 36 zonas, tendo a Microsoft perspetivas para incorporar mais 6 zonas [18, 27].

2.3.4.4 Serviços para o *deployment* e execução de micro-serviços

Os *containers* tendo vários benefícios, como os enumerados na secção 2.2.3.4, são a forma indicada para o *deployment* e execução de micro-serviços. Tendo em conta a análise efetuada em relação aos serviços fornecidos pela Amazon, Microsoft e Google, a escolha da infraestrutura poderia recair sobre qualquer uma delas, sendo que a Amazon e a Microsoft fornecem mais opções como o Docker e o Kubernetes, e a Google apenas disponibiliza o Kubernetes.

2.4 *Edge Computing*

Embora a computação em *Cloud* permita simplificar o processamento e armazenamento dos dados através de uma infraestrutura altamente disponível e com grande capacidade, toda a computação ocorre de uma forma centralizada nos servidores dos fornecedores de serviços (*cloud*). Tal implica que, na realização de operações, os clientes necessitam sempre de comunicar com os servidores. Este requisito leva a que haja um aumento das comunicações entre os dispositivos dos utilizadores e os servidores, os quais poderão não ter tempos de resposta adequados, o que é mais grave no caso de aplicações que requeiram uma baixa latência [54].

A computação na *Edge* (*Edge Computing*) vem possibilitar que, parte da computação necessária a uma aplicação seja realizada na periferia da rede, ou seja, em dispositivos mais próximos dos dispositivos dos utilizadores, *edge nodes*, como por exemplo, *routers*, *switches* e *base stations*. Neste contexto os dispositivos para além de consumirem os dados, também os produzem, ou seja, estes pedem recursos e serviços à *cloud*, mas também realizam computação para a *cloud* [50].

A Figura 2.6 representa onde se localiza a computação na *Edge*.

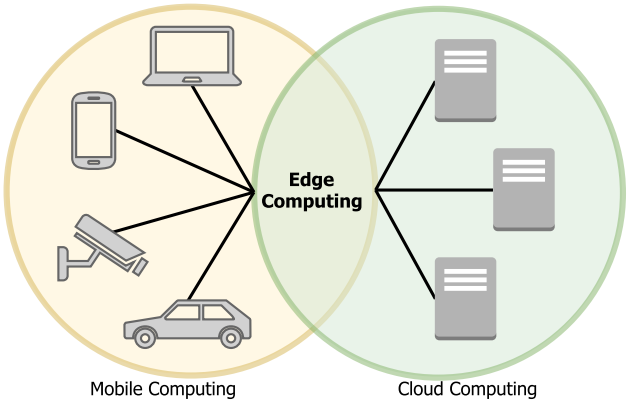

Figura 2.6: Localização da computação na *Edge*. Adaptado de [6].

2.4.1 Benefícios da *Edge Computing*

Realizar o processamento na *edge* permite certas vantagens em relação à computação em *Cloud*, sendo que as principais são [50, 54]:

- Diminuir os tempos de latência de certas operações, resultando numa melhor experiência de utilização para o utilizador;
- Processar dados na *edge*, enviando o resultado para a *cloud*, diminuindo assim o volume de dados transferidos entre o utilizador e a *cloud*;
- Reduzir a carga nos servidores na *cloud*, de forma a que fiquem mais disponíveis para outra tarefas e também para que o consumo de energia seja menor.

2.4.2 Desafios da *Edge Computing*

Devido a ser um tipo de computação relativamente recente, existem ainda certos desafios que necessitam de alguma atenção.

- **Computação nos *edge nodes***: a computação pode ser executada em diversos nós que estão localizados entre o dispositivo na *edge* e a *cloud*, mas nem todos estes nós são indicados para certos tipos de computação como, por exemplo, as *base stations* possuem processadores de sinal digital (*Digital Signal Processors* - DSPs), que não estão indicados para a computação geral [54].

  Esta questão poderá vir a ser ultrapassada, por exemplo, nos routers wireless, ao serem atualizados para terem mais capacidade de processamento, mas por enquanto tal incorre em custos bastante elevados [26, 54].

- **Programação**: a *cloud* permite ao programador um nível de abstração em que o código desenvolvido será *deployed* em máquinas semelhantes. Na *edge computing* existe uma heterogeneidade nos dispositivos, dificultando assim como o código é desenvolvido, tendo que ser adequado a cada tipo de dispositivo/plataforma [50].

- **Descoberta de *edge nodes***: com a existência de inúmeros dispositivos na *edge*, é necessário realizar uma filtragem de forma a que apenas sejam utilizados os dispositivos que preencham os requisitos necessários, sendo que esta descoberta não pode ser manual devido ao grande número de dispositivos [54].

- **Privacidade e segurança**: são duas características bastante relevantes em qualquer sistema, e quando se fala em computação na *Edge* não é diferente. Por exemplo, ao utilizar um *router* para processar certos dados, este não pode acedido por um utilizador mal-intencionado, possibilitando que este veja dados privados [54].

  A inexistência de mecanismos eficientes de proteção dos dados na *edge* é um problema, sendo que o seu ambiente bastante dinâmico faz com que esta rede seja mais vulnerável. Alguns dispositivos possuem capacidades bastante limitadas, fazendo com que os métodos de segurança utilizados atualmente possam não ser os mais adequados [50].

2.5 Computação na *Cloud* e *Edge*

A computação em plataformas heterogéneas, composta pela *cloud* e *edge*, com elevado número de recursos disponíveis na *edge*, levanta assim um conjunto de desafios em termos do desenvolvimento e execução das aplicações. Estes desafios incluem a localização adequada dos serviços e a sua coordenação, o seu *deployment*, dependendo dos recursos dos nós computacionais, o volume de acessos por parte dos utilizadores, como garantir questões sobre segurança e privacidade, que modelos de custos são vantajosos tanto para os fornecedores de serviços como para os utilizadores, etc.

A computação osmótica (*Osmotic Computing*) [55] é um dos paradigmas recentes que pretende dar resposta a esses desafios. O seu objetivo é permitir um *deploy* automático de micro-serviços tanto na *edge* como na *cloud* através de uma gestão eficiente dos recursos disponíveis, obtendo uma melhor capacidade computacional mais perto do utilizador [48, 55].

Este paradigma implica uma gestão dinâmica dos serviços na *cloud* e na *edge*, tendo em conta possíveis problemas relacionados com o *deployment*, rede e segurança.

A computação osmótica realça os novos desafios resultantes da heterogeneidade dos recursos da *cloud* e da *edge*, bem como os relacionados com a elasticidade da *cloud*.

Na perspetiva deste paradigma, os micro-serviços são instalados em *containers* e *deployed* na *cloud* ou na *edge*, bem como a migração dos micro-serviços deve ser realizada de forma dinâmica e eficiente para evitar problemas, como paragens ou degradação da aplicação [55].

2.5.1 Monitorização em arquiteturas heterogéneas

Muitas das dificuldades em monitorizar a atividade de um sistema heterogéneo com estas características prendem-se com a escala e complexidade da infraestrutura considerada como alvo do *deployment* de micro-serviços, tal como realçado no paradigma da computação osmótica [55].

A infraestrutura ao incluir recursos de hardware na *cloud*, *edge* e na ligação entre ambas, bem como a heterogeneidade e escala dos micro-serviços que compõem as aplicações, resulta num sistema complexo cuja deteção de problemas (ex.: alta latência em pedidos) e identificação da fonte desses problemas, se torna bastante difícil. Para mais, é necessário monitorizar a utilização de recursos na infraestrutura de modo a garantir a Qualidade de Serviço contratada com os utilizadores (ex.: recorrendo aos mecanismos de escalabilidade disponíveis).

Torna-se assim difícil implementar técnicas eficientes de monitorização que suportem a resolução de problemas, contribuam para a coordenação dos micro-serviços e sua qualidade de serviço quando, por exemplo, os micro-serviços têm de ser migrados para a periferia e o seu estado tem de continuar a ser monitorizado. Esta problemática é tratada numa outra dissertação de mestrado que, conjuntamente com esta, contribui para o objetivo geral da gestão automática de micro-serviços nestas infraestruturas heterogéneas.

2.5.2 Orquestração de micro-serviços e controlo da elasticidade

Num sistema *edge* e *cloud*, a orquestração dos micro-serviços é um problema complexo devido à dificuldade em estimar o comportamento da carga de trabalho em termos de volume de dados a serem processados, taxa de chegada de dados, tipos de *queries*, número de utilizadores conectados a diferentes tipos de micro-serviços, etc. Sem determinadas informações sobre os micro-serviços, como a carga de trabalho de cada um deles, é difícil saber qual a escala em termos de recursos que deve ser escolhida para os micro-serviços [55].

Por exemplo, o Kubernetes permite a reconfiguração dos *containers* que contêm os micro-serviços, escalando de forma automática através da análise da utilização de CPU.

A computação nestas plataformas heterogéneas deve assim permitir o controlo e a reconfiguração em tempo de execução dos recursos disponíveis na *edge*, para além dos já considerados na *cloud*.

2.5.3 Migração de serviços

Em ambientes de *cloud*, a migração pode permitir que seja realizada uma gestão mais eficiente dos recursos. Esta consiste no processo de mover VMs entre diferentes máquinas físicas, podendo ser utilizada para atingir vários objetivos, como a redução de energia, balanceamento de carga e tolerância a falhas [51].

2.5.3.1 Métodos de migração

A migração pode ser classificada em migração *non-live* e migração *live*. Na migração *non-live*, primeiro a VM é colocada em suspensão antes de se iniciar o processo de transferência para outro

local, e depois é novamente colocada em execução quando as tarefas de migração terminarem. Durante este tipo de migração, os serviços que sejam inseridos na VM não se encontram acessíveis.

A migração *live* é uma técnica mais avançada, não necessitando que as VMs sejam suspensas. Este tipo de migração tem de ser rápida e eficiente, de forma a que a virtualização permita um balanceamento da carga dinâmico e a manutenção sem tempos de inatividade.

2.5.3.2 Objetivos da migração *live*

Através da utilização desta técnica de migração, pode ser possível melhorar o desempenho, isto se for utilizada a classificação adequada. Alguns exemplos são:

1. **Consolidação do servidor**: a gestão da energia pode ser conseguida através da consolidação do servidor, isto é, otimizar o número de máquinas físicas utilizadas diminuindo o seu número total. Esta deve ser utilizada quando existe um grande número de máquinas físicas subutilizadas, devendo proceder-se à migração de VMs de máquinas físicas menos sobrecarregadas para outras mais sobrecarregadas.
2. **Balanceamento de carga**: permite diminuir o desequilíbrio ao nível de utilização dos recursos das máquinas físicas. A carga total do sistema pode ser ajustada através da recolocação de VMs de máquinas físicas sobrecarregadas para outras com menos carga.

2.5.3.3 Métricas de migração *live*

De forma a avaliar o desempenho da migração *live* são consideradas as seguintes métricas: tempo de inatividade, tempo de migração total e a quantidade de dados na migração. Em relação à avaliação em termos de consumo energético são utilizadas as métricas: consumo do recurso, utilização do recurso e número de VMs por servidor.

Podem ainda ser consideradas, consoante o tipo de serviço:

- Taxa de transferência: utilizada para calcular o número de tarefas executadas.
- Tempo de resposta: tempo de resposta de um dado algoritmo de balanceamento de carga.
- Largura de banda: a capacidade de rede disponível durante a migração.

A identificação de como os micro-serviços podem ser migrados da *cloud* para a *edge* (e ao contrário), bem como os fatores que influenciam essa migração ainda são um desafio [48, 55].

2.6 Sumário

Ao longo deste capítulo foi realizada uma investigação sobre os conceitos, abordagens e tecnologias relevantes para o desenvolvimento da solução proposta.

Os micro-serviços são uma arquitetura bastante versátil, pois ao serem compostos por serviços de pequenas dimensões e independentes, permite que estes escalem de forma independente e que o seu *deployment* seja realizado tanto em infraestruturas na *cloud* como na *edge*.

Foi ainda possível obter mais conhecimentos sobre propostas de soluções já existentes para a resolução dos problemas identificados (na secção 1.2), como a computação osmótica cujo objetivo é permitir o *deployment* automático dos micro-serviços quer na *cloud* quer na *edge*.

No capítulo seguinte será abordada em mais detalhe a solução proposta em 1.3, que irá tratar dos problemas apresentados em 1.2.

Capítulo 3

Solução proposta

O presente capítulo tem como objetivo apresentar a solução proposta. Na secção 3.1 é apresentada a descrição da solução que será implementada, nomeadamente que funcionalidades irão ser fornecidas. Na secção 3.2 são referidas quais os passos que irão ser seguidos para o desenvolvimento da solução, bem como quais as ferramentas que poderão vir a ser utilizadas. Para finalizar, a secção 3.3 apresenta e descreve o plano de trabalho que irá ser seguido para a realização da dissertação.

3.1 Descrição geral da solução proposta

A solução compreende o desenvolvimento de mecanismos e funcionalidades para aplicações com uma arquitetura de micro-serviços que permitam a gestão e coordenação na migração dos serviços em termos do seu local de *deployment*, mais precisamente entre a *cloud* e a *edge*. Estes mecanismos serão fornecidos através do desenvolvimento de um componente responsável pela migração automática dos micro-serviços que compõem a aplicação.

A solução terá como base inicial o estudo de uma aplicação baseada em micro-serviços denominada *Sock Shop* [57], a qual é composta por 6 micro-serviços:

1. **Payment**: fornece serviços de pagamento. Desenvolvido em Go.
2. **Orders**: permite processamento de encomendas. Desenvolvido em Java e base de dados em MongoDB.
3. **Carts**: fornece carrinhos de compras aos utilizadores. Desenvolvido em Java e base de dados em MongoDB.
4. **Catalogue**: fornece informações dos produtos/catálogos. Desenvolvido em Go e base de dados em MySQL.
5. **Users**: contém as informações sobre a conta do utilizador, como cartões e endereços. Desenvolvido em Go e base de dados em MongoDB.
6. **Shipping**: fornece funcionalidades de expedição. Desenvolvido em Java.

Será ainda desenvolvida uma funcionalidade que permite configurar quais as regras e métricas que o componente de migração terá em conta na sua decisão (com possibilidade de alteração). Por exemplo, configurar a partir de que tempo de latência um micro-serviço deve ser migrado para a *edge*. Esta funcionalidade estará incorporada no componente de migração, e terá uma interface de administração, de forma a que um administrador possa fazer as alterações pretendidas. Os valores que o componente de migração analisa são enviados por um outro componente que é responsável pela monitorização dos micro-serviços, e dependem também de informação prévia comunicada por este mesmo componente de monitorização.

3.2 Metodologias e ferramentas

A solução será desenvolvida de forma incremental, sendo que inicialmente será considerado que um micro-serviço, quando exista necessidade de o migrar, não tem dependências de outro micro-serviços. Na iteração seguinte, o componente de migração já terá em conta as dependências de um dado serviço que tenha de ser migrado, sendo que neste caso os outros serviços poderão no extremo ter que ser migrados também. Será inclusive considerada a possibilidade de serem alteradas dinamicamente as regras e métricas sobre as quais o componente de migração opera, quando decide se deve migrar ou não um micro-serviço.

Os micro-serviços que compõem a aplicação usada como caso de estudo inicial (*Sock Shop*) estão desenvolvidos em Go e em Java (utilizando a *framework* Spring). Espera-se que venha também a ser utilizada a *framework* Spring para o desenvolvimento da versão inicial do componente de migração, sendo que a interface de utilizador que permite a alteração dinâmica das regras será desenvolvida com tecnologias Web (HTML, Javascript).

O *deployment* dos micro-serviços e do componente de migração deverá ser realizado com recurso a *containers*, sendo que serão considerados para tal o Kubernetes e/ou o Docker.

Em relação à avaliação da solução, esta será realizada tendo em conta as seguintes métricas:

- **Tempo de resposta**: o tempo de resposta das operações, quando existe a migração de serviços.
- **Custo de migração**: os recursos que o componente de migração requer quando está em inativo, e quando está em tarefas de migração, sendo consideradas a taxa de utilização de CPU e a quantidade de RAM.
- **Tempo de migração**: o tempo que demora a migração de um serviço a ser realizada.

3.3 Plano de trabalho

O plano de trabalho que será seguido encontra-se representado na Figura 3.1, sendo decomposto nas seguintes tarefas:

1. **Análise da arquitetura da aplicação**: analisar as dependências entre os micro-serviços, quais as ferramentas, metodologias e tecnologias utilizadas e a utilizar. Avaliar quais as infraestruturas de *cloud* indicadas para o *deploy* dos micro-serviços.
2. **Estudar se existe a necessidade de adição de novos micro-serviços**: verificar se é necessário desenvolver um novo micro-serviço, para integrar uma dada funcionalidade, de modo a tornar a aplicação mais complexa e com mais oportunidades de migração.

3. **Desenvolvimento do sistema, com gestão e coordenação da migração automática de micro-serviços**: que se decompõe nas seguintes tarefas.

   3.1. **Identificar quais os dispositivos na *edge* que podem ser utilizados**: identificar e selecionar quais os dispositivos na *edge* capazes de executar os micro-serviços.

   3.2. **Avaliar as hipóteses de migração dos micro-serviços**: replicação dos micro-serviços (escalabilidade horizontal, réplicas), replicar na *cloud*, ou migrar para *edge* (um ou mais nós).

   3.3. **Configuração de métricas estáticas para a migração**: definir e implementar as métricas e regras relevantes na migração ou replicação de forma estática.

   3.4. **Implementação do componente que permita a migração e replicação**: desenvolvimento do componente que permite a migração e replicação dos serviços.

   3.5. **Permitir uma alteração dos cenários de forma dinâmica**: possibilitar a alteração, a adição ou remoção de regras que são utilizadas para a decisão da migração ou replicação.

4. **Avaliação e melhorias do sistema**: realização de testes ao sistema. Caso seja necessário serão corrigidos problemas e feitas melhorias ao sistema.

5. **Elaboração do documento da dissertação**: redação da dissertação, incluindo o trabalho desenvolvido, análises críticas com base nos resultados obtidos e conclusões finais.

Nome da Tarefa	Duração	Início	Conclusão
Aplicação com gestão e coordenação da migração automática de micro-serviços	196 dias	Seg 05/03/18	Dom 16/09/18
1 Análise da arquitetura da aplicação	7 dias	Seg 05/03/18	Dom 11/03/18
2 Estudar se existe a necessidade de adição de novos micro-serviços	7 dias	Seg 12/03/18	Dom 18/03/18
3 Desenvolvimento do sistema, com gestão e coordenação da migração automática de micro-serviços	126 dias	Seg 19/03/18	Dom 22/07/18
3.1 Identificar quais os dispositivos na edge que podem ser utilizados	7 dias	Seg 19/03/18	Dom 25/03/18
3.2 Avaliar as hipóteses de migração dos micro-serviços	21 dias	Seg 26/03/18	Dom 15/04/18
3.3 Configuração de métricas estáticas para a migração	21 dias	Seg 16/04/18	Dom 06/05/18
3.4 Implementação do componente que permita a migração e replicação	63 dias	Seg 07/05/18	Dom 08/07/18
3.5 Permitir uma alteração dos cenários de forma dinâmica	14 dias	Seg 09/07/18	Dom 22/07/18
4 Avaliação e melhorias do sistema	21 dias	Seg 23/07/18	Dom 12/08/18
5 Elaboração do documento da dissertação	56 dias	Seg 23/07/18	Dom 16/09/18

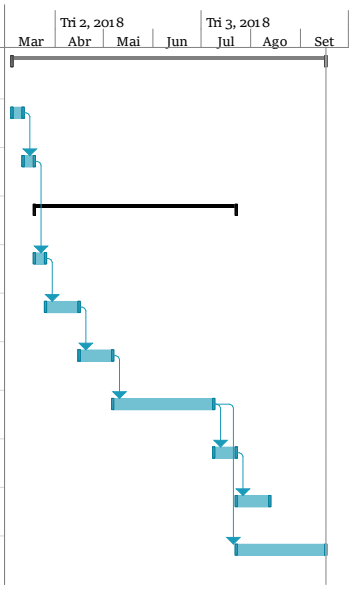

Figura 3.1: Gantt chart.

Bibliografia

[1] Amazon. *AWS Global Infrastructure*. https://aws.amazon.com/about-aws/global-infrastructure/. Consultado: Jan. de 2018.

[2] P. Bak, R. Melamed, D. Moshkovich, Y. Nardi, H. J. Ship e A. Yaeli. "Location and Context-Based Microservices for Mobile and Internet of Things Workloads". Em: *2015 IEEE International Conference on Mobile Services, MS 2015, New York City, NY, USA, June 27 - July 2, 2015*. 2015, pp. 1–8. DOI: 10.1109/MobServ.2015.11. URL: https://doi.org/10.1109/MobServ.2015.11.

[3] A. Balalaie, A. Heydarnoori e P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". Em: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: 10.1109/MS.2016.64. URL: https://doi.org/10.1109/MS.2016.64.

[4] M. Champion. *How We Built Our Stack For Shipping at Scale*. Ed. por HubSpot. https://product.hubspot.com/blog/how-we-built-our-stack-for-shipping-at-scale. Consultado: Jan. de 2018.

[5] L. Columbus. *Forrester's 10 Cloud Computing Predictions For 2018*. Ed. por Forbes. https://www.forbes.com/sites/louiscolumbus/2017/11/07/forresters-10-cloud-computing-predictions-for-2018. Consultado: Jan. de 2018.

[6] O. E. Computing. *Open Edge Computing*. http://openedgecomputing.org/. Consultado: Jan. de 2018.

[7] Confluent, Inc. *Building a Microservices Ecosystem with Kafka Streams and KSQL*. https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/. Consultado: Jan. de 2018.

[8] F. Curbera, R. Khalaf, N. Mukhi, S. Tai e S. Weerawarana. "The Next Step in Web Services". Em: *Commun. ACM* 46.10 (out. de 2003), pp. 29–34. ISSN: 0001-0782. DOI: 10.1145/944217.944234. URL: http://doi.acm.org/10.1145/944217.944234.

[9] Docker. *Package software into standardized units for development, shipment and deployment*. https://www.docker.com/what-container#package_software. Consultado: Dez. de 2017.

[10] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin e L. Safina. "Microservices: yesterday, today, and tomorrow". Em: *CoRR* abs/ 1606.04036 (2016). arXiv: 1606.04036. URL: http://arxiv.org/abs/1606.04036.

[11] R. T. Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[12] I. T. Foster, Y. Zhao, I. Raicu e S. Lu. "Cloud Computing and Grid Computing 360-Degree Compared". Em: *CoRR* abs/0901.0131 (2009). arXiv: 0901.0131. URL: http://arxiv.org/abs/0901.0131.

[13] M. Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. Consultado: Set. de 2017. Mar. de 2014.

[14] M. Fowler. *Microservice Trade-Offs*. https://martinfowler.com/articles/microservice-trade-offs.html. Consultado: Nov. de 2017. Jul. de 2015.

[15] Gajotres.net. *Top 8 Java RESTful Micro Frameworks – Pros/Cons*. https://www.gajotres.net/best-available-java-restful-micro-frameworks/. Consultado: Jan. de 2018.

[16] M. Garriga. "Towards a Taxonomy of Microservices Architectures". Em: (2017).

[17] M. Garriga, C. Mateos, A. Flores, A. Cechich e A. Zunino. "RESTful service composition at a glance: A survey". Em: *Journal of Network and Computer Applications* 60.Supplement C (2016), pp. 32 –53. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.11.020. URL: http://www.sciencedirect.com/science/article/pii/S1084804515002933.

[18] Google. *Cloud Locations*. https://cloud.google.com/about/locations/. Consultado: Jan. de 2018.

[19] C. Harvey e A. Patrizio. *AWS vs. Azure vs. Google: Cloud Comparison*. Ed. por Datamation. https://www.datamation.com/cloud-computing/aws-vs.-azure-vs.-google-cloud-comparison.html. Consultado: Jan. de 2018.

[20] S. Hassan e R. Bahsoon. "Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap". Em: *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. 2016, pp. 813–818. DOI: 10.1109/SCC.2016.113. URL: https://doi.org/10.1109/SCC.2016.113.

[21] John B. *Using Services to Break Down Monoliths*. Ed. por Yelp. https://engineeringblog.yelp.com/2015/03/using-services-to-break-down-monoliths.html. Consultado: Jan. de 2018.

[22] N. Kratzke e P. Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". Em: *Journal of Systems and Software* 126 (2017), pp. 1–16. DOI: 10.1016/j.jss.2017.01.001. URL: https://doi.org/10.1016/j.jss.2017.01.001.

[23] D. C. Marinescu. "Chapter 5 - Cloud Resource Virtualization". Em: *Cloud Computing*. Ed. por D. C. "Marinescu. Boston: Morgan Kaufmann, 2013, pp. 131 –161. ISBN: 978-0-12-404627-6. DOI: https://doi.org/10.1016/B978-0-12-404627-6.00005-1. URL: https://www.sciencedirect.com/science/article/pii/B9780124046276000051.

[24] D. C. Marinescu. "Chapter 6 - Cloud Resource Management and Scheduling". Em: *Cloud Computing*. Ed. por D. C. Marinescu. Boston: Morgan Kaufmann, 2013, pp. 163 –203. ISBN: 978-0-12-404627-6. DOI: https://doi.org/10.1016/B978-0-12-404627-6.00006-3. URL: https://www.sciencedirect.com/science/article/pii/B9780124046276000063.

[25] P. M. Mell e T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Rel. téc. Gaithersburg, MD, United States, 2011.

[26] C. Meurisch, A. Seeliger, B. Schmidt, I. Schweizer, F. Kaup e M. Mühlhäuser. "Upgrading Wireless Home Routers for Enabling Large-Scale Deployment of Cloudlets". Em: *Mobile Computing, Applications, and Services - 7th International Conference, MobiCASE 2015, Berlin, Germany, November 12-13, 2015, Revised Selected Papers*. 2015, pp. 12–29. DOI: 10.1007/978-3-319-29003-4_2. URL: https://doi.org/10.1007/978-3-319-29003-4_2.

[27] Microsoft. *Azure regions*. https://azure.microsoft.com/en-us/regions/. Consultado: Jan. de 2018.

[28] S. Mumbaikar, P. Padiya et al. "Web services based on soap and rest principles". Em: *International Journal of Scientific and Research Publications* 3.5 (2013), pp. 1–4.

[29] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. 1ª ed. O'Reilly Media, Inc., p. 282. ISBN: 978-1491950357.

[30] NGINX. *What Is Load Balancing? How Load Balancers Work*. https://www.nginx.com/resources/glossary/load-balancing/. Consultado: Jan. de 2018.

[31] M. P. Papazoglou e D. Georgakopoulos. "Introduction: Service-oriented Computing". Em: *Commun. ACM* 46.10 (out. de 2003), pp. 24–28. ISSN: 0001-0782. DOI: 10.1145/944217.944233. URL: http://doi.acm.org/10.1145/944217.944233.

[32] M. P. Papazoglou, P. Traverso, S. Dustdar e F. Leymann. "Service-Oriented Computing: State of the Art and Research Challenges". Em: *Computer* 40.11 (2007), pp. 38–45. ISSN: 0018-9162. DOI: 10.1109/MC.2007.400.

[33] C. Pautasso. *SOAP vs. REST Bringing the Web back into Web Services*. Ed. por IBM Zurich Research Lab. 2007.

[34] *Raw benchmarks on throughput, latency and transfer of Hello World on popular microservices frameworks*. https://github.com/networknt/microservices-framework-benchmark. Consultado: Jan. de 2018.

[35] M. Richards. *Microservices vs. Service-Oriented Architecture*. 1ª ed. O'Reilly Media, Inc., p. 44. ISBN: 978-1-491-95242-9.

[36] C. Richardson. *Pattern: API Gateway / Backend for Front-End*. http://microservices.io/patterns/apigateway.html. Consultado: Nov. de 2017.

[37] C. Richardson. *Pattern: Client-side service discovery*. http://microservices.io/patterns/client-side-discovery.html. Consultado: Dez. de 2017.

[38] C. Richardson. *Pattern: Event sourcing*. http://microservices.io/patterns/data/event-sourcing.html. Consultado: Dez. de 2017.

[39] C. Richardson. *Pattern: Microservice Architecture*. http://microservices.io/patterns/microservices.html. Consultado: Jan. de 2018.

[40] C. Richardson. *Pattern: Monolithic Architecture*. http://microservices.io/patterns/monolithic.html. Consultado: Jan. de 2018.

[41] C. Richardson. *Pattern: Multiple service instances per host*. http://microservices.io/patterns/deployment/multiple-services-per-host.html. Consultado: Dez. de 2017.

[42] C. Richardson. *Pattern: Publish events using database triggers*. http://microservices.io/patterns/data/database-triggers.html. Consultado: Dez. de 2017.

[43] C. Richardson. *Pattern: Server-side service discovery*. http://microservices.io/patterns/server-side-discovery.html. Consultado: Dez. de 2017.

[44] C. Richardson. *Pattern: Service instance per container*. http://microservices.io/patterns/deployment/service-per-container.html. Consultado: Dez. de 2017.

[45] C. Richardson. *Pattern: Service Instance per VM*. http://microservices.io/patterns/deployment/service-per-vm.html. Consultado: Dez. de 2017.

[46] C. Richardson. *Pattern: Service registry*. http://microservices.io/patterns/service-registry.html. Consultado: Dez. de 2017.

[47] C. Richardson e F. Smith. *Microservices: From Design to Deployment*. Ed. por NGINX. 2016.

[48] V. Sharma, K. Srinivasan, D. N. K. Jayakody, O. F. Rana e R. Kumar. "Managing Service-Heterogeneity using Osmotic Computing". Em: *CoRR* abs/1704.04213 (2017). arXiv: 1704.04213. URL: http://arxiv.org/abs/1704.04213.

[49] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne e X. Xu. "Web services composition: A decade's overview". Em: *Information Sciences* 280.Supplement C (2014), pp. 218 –238. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2014.04.054. URL: http://www.sciencedirect.com/science/article/pii/S0020025514005428.

[50] W. Shi, J. Cao, Q. Zhang, Y. Li e L. Xu. "Edge Computing: Vision and Challenges". Em: *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.

[51] A. Son e E. Huh. "Migration Method for Seamless Service in Cloud Computing: Survey and Research Challenges". Em: *30th International Conference on Advanced Information Networking and Applications Workshops, AINA 2016 Workshops, Crans-Montana, Switzerland, March 23-25, 2016*. 2016, pp. 404–409. DOI: 10.1109/WAINA.2016.72. URL: https://doi.org/10.1109/WAINA.2016.72.

[52] Techopedia. *Virtual Machine Monitor (VMM)*. https://www.techopedia.com/definition/717/virtual-machine-monitor-vmm. Consultado: Jan. de 2018.

[53] The Apache Software Foundation. *Apache ZooKeeper*. https://zookeeper.apache.org/. Consultado: Jan. de 2018.

[54] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick e D. S. Nikolopoulos. "Challenges and Opportunities in Edge Computing". Em: *2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016*. 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18. URL: https://doi.org/10.1109/SmartCloud.2016.18.

[55] M. Villari, M. Fazio, S. Dustdar, O. F. Rana e R. Ranjan. "Osmotic Computing: A New Paradigm for Edge/Cloud Integration". Em: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83. DOI: 10.1109/MCC.2016.124. URL: https://doi.org/10.1109/MCC.2016.124.

[56] W. Vogels. "Eventually Consistent". Em: *ACM Queue* 6.6 (2008), pp. 14–19. DOI: 10.1145/1466443.1466448. URL: http://doi.acm.org/10.1145/1466443.1466448.

[57] Weaveworks, Inc. *Sock Shop*. https://microservices-demo.github.io/. Consultado: Jan. de 2018. 2017.

[58] D. Wolf. *Who wins the three-way cloud battle? Google vs. Azure vs. AWS*. Ed. por ReadWrite. https://readwrite.com/2017/02/20/wins-three-way-cloud-battle-google-vs-azure-vs-aws-dl1/. Consultado: Jan. de 2018.

[59] E. Yanaga. *Migrating to Microservice Databases: From Relational Monolith to Distributed Data*. 1ª ed. O'Reilly Media, Inc., p. 60. ISBN: 978-1-491-97461-2. URL: https://developers.redhat.com/promotions/migrating-to-microservice-databases/.

[60] Q. Zhang, L. Cheng e R. Boutaba. "Cloud computing: state-of-the-art and research challenges". Em: *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.