Daniel Filipe Santos Pimenta

Licenciado em Engenharia Informática

Gestão Dinâmica de Micro-serviços na Cloud/Edge

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

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

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

Júri

Presidente: Prof. Drª Susana Maria dos Santos Nascimento Martins de Almeida, FCT-NOVA

Arguente: Prof. Dr. João Nuno de Oliveira e Silva, IST-UL

Vogal: Prof. Drª Maria Cecília Farias Lorga Gomes, FCT-NOVA

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2021

Gestão Dinâmica de Micro-serviços na Cloud/Edge

Aos meus pais, irmãos e avós.

Agradecimentos

Um muito obrigado à minha orientadora, a Professora Doutora Maria Cecília Farias Lorga Gomes, por ter sido paciente na demora da conclusão, e pelos conselhos e disponibilidade durante a realização da minha dissertação. Agredecimento ao meu co-orientador João Carlos Antunes Leitão, pelos conselhos e sabedoria transmitida que me ajudaram a resolver problemas durante a dissertação.

A todos os professores da Faculdade de Ciências e Tecnologia com quem tive o prazer de aprender os conhecimentos que tornaram possível esta dissertação.

Aos meus pais e avós por apoiarem os meus estudos.

Resumo

Observa-se, hoje em dia, um crescimento muito elevado da utilização de dispositivos no domínio da Internet of Things (IoT) e de dispositivos móveis, bem como do número de aplicações com consumo elevado de largura de banda (ex.: visualização de vídeos, a pedido). Tal implica que, num futuro próximo, não será viável suportar a quantidade de dados transferidos entre os dispositivos clientes ("*end devices*") e os centros de dados *cloud*, onde tipicamente são alojadas aplicações de acesso ubíquo. O problema do consequente aumento da latência percebido nas aplicações clientes, é ainda agravado no caso de aplicações "sensíveis à latência" *(latency sensitive)*, como sejam aplicações bastante interativas ou de tempo real/quase-real (ex.: carros autonómicos, jogos online, etc.). A localização deste tipo de aplicações na *cloud*, onde é grande a distância entre os clientes e a localização dos centros de dados, resulta em níveis inaceitáveis de Quality of Service (QoS) percebida pelos clientes, ou mesmo a impossibilidade de cumprir os requisitos funcionais das aplicações. A computação na *edge* (*Edge computing*) surge como resposta aos problemas de latência referidos, ao usar recursos computacionais dos dispositivos na periferia da rede, que se situam mais próximo das aplicações cliente. É ainda possível realizar computações que filtrem os dados gerados na periferia, contribuindo para diminuir o volume de dados em trânsito, e que teriam de ser processados na *cloud*. Comparativamente com os recursos presentes nos centros de dados *cloud*, os recursos dos nós na *edge* são, no entanto, de capacidade computacional bastante limitada. Isto implica que utilizar aplicações monolíticas (tipicamente de grandes dimensões) não é uma opção eficaz na computação na *edge*, quer pelo custo da sua migração/replicação, quer pela impossibilidade de alojar as aplicações nesses nós. O uso da arquitetura de micro-serviços permite contribuir para a resolução deste problema. As aplicações são compostas por múltiplos micro-serviços, cada um com pequena dimensão, oferecendo uma funcionalidade única, com interfaces bem definidas e que comunicam entre si através de mensagens, tornando-os independentes entre si. Desta forma, é possível realizar uma gestão mais eficaz dos recursos disponíveis nos nós periféricos.

O trabalho que se procura resolver relaciona-se com os problemas inerentes a um gestor centralizado quando temos um domínio muito dinâmico, o que inclui quer a infraestrutura, quer as aplicações variadas que são lançadas nesses nós, quer a grande volatilidade e diversidade nos acessos por parte dos clientes. Foi desenvolvida uma solução

de um gestor distribuído, com arquitetura hierárquica, de um conjunto de nós e serviços no ambiente *cloud*/*edge*, incluindo a necessidade de efetuar melhoramentos adicionais à solução existente. A hierarquia de gestores distribuídos permitiu distribuir a responsabilidade e trabalho, ao reduzir a região geográfica abrangida por cada gestor, reduzindo assim a distância de comunicação entre os nós que cada um controla.

Palavras-chave: *Cloud*, *Edge*, Micro-serviço, Gestão de recursos, Virtualização, Sistema distribuído

Abstract

There is nowadays a very high growth in the use of devices in the domain of IoT and mobile devices, as well as the number of applications with high bandwidth consumption (e.g. video on-demand). This implies that in the near future it will not be feasible to support the amount of data transferred between end devices and data centers, where ubiquitous access applications are typically hosted. The problem of the consequent increase in perceived latency in client applications is further aggravated in the case of latency-sensitive applications, such as very interactive or real-time applications (e.g. autonomic cars, online games, etc.). The location of this type of cloud applications, where the distance between customers and the location of data centers is large, results in unacceptable levels of QoS perceived by customers, or even the impossibility of meeting the functional requirements of such applications.

Edge computing emerges as a response to the latency problems referred to by using computing resources of the devices at the edge of the network, which are closer to the client applications. It is also possible to perform computations that filter the data generated in the edge, contributing to decrease the volume of data in transit, that would have to be otherwise processed in the cloud. Compared with the resources available in the cloud data centers, the resources of the nodes in the edge are, however, of very limited computational capacity. This implies that using monolithic (typically large) applications is not an efficient choice in edge computing, either because of the cost of its migration/replication or because it is impossible to host the applications on those nodes. The use of the micro-services architecture seeks to solve this problem. The applications are composed of multiple micro-services, each with a small dimension, offering a unique functionality, with well-defined interfaces that communicate with each other through messages, making them independent of each other. In this way, it is possible to perform a more efficient management of the available resources in the edge devices.

This work looks to solve problems related to a centralized manager on a very dynamic domain, including the infrastructure, multiple heterogeneous aplications launched on the system nodes, and the diversity and volatibility of the users. The prototype consists of a hierarchy of distributed managers of nodes and services in the cloud/edge environment, including the necessity of doing aditional improvements of the existing work. The hierarchy of distributed managers allows the distribution of work and resposabilities between

managers and the reduction of the geographical area managed by one particular manager, which implies a lesser distance of communication between nodes.

Keywords: Cloud, Edge, Microservice, Resource Management, Virtualization, Distributed system

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

aplicação monolítica	Uma aplicação monolítica não permite a execução independente dos seus módulos.
economia de escala	Economias de escala são os fatores que conduzem à redução do custo médio de produção de um determinado bem, à medida que a quantidade produzida aumenta.
latência	Período de latência é a diferença de tempo entre o início de um evento e o momento em que os seus efeitos se tornam percetíveis.
on-premises	*On-premises software*, também conhecido como *shrink wrap*, é um modelo de distribuição de software que é instalado e operado numa infraestrutura computacional presente no local do cliente.
osmose	Osmose é o processo de difusão de moléculas, de um maior para um menor ponto de concentração.

Siglas

AMI	Amazon Machine Images
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APIs	Application programming interfaces
AWS	Amazon Web Services
CDN	Content Delivery Network
CoAP	Constrained Application Protocol
CORS	Cross-origin resource sharing
CPU	Central Process Unit
CSS	Cascading Style Sheets
CSV	Comma-separated values
DDoS	Distributed Denial of Service
DDS	Data Distribution Service
DTO	Data Transfer Object
ESB	Enterprise Service Bus
GCE	Google Compute Engine
GCP	Google Cloud Platform
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IaaS	Infrastructure as a Service
IDE	Integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol

JAR	Java ARchive
JPA	Java Persistence API
JSON	JavaScript Object Notation
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
P2P	Peer-to-Peer
PaaS	Platform as a Service
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SFTP	SSH File Transfer Protocol
SLA	Service Level Agreement
SLOs	Service Level Objectives
SSH	Secure Shell
SVG	Scalable Vector Graphics
URL	Uniform Resource Locator
vCPU	Virtual Centralized Processing Unit
VM	Virtual Machine
VOD	Video on Demand
XML	Extensible Markup Language

Símbolos

Capítulo 1

Introdução

1.1 Contexto e motivação

O paradigma de computação centralizado, denominado por computação cloud (*cloud computing*), tem sido o mais predominante, com maior interesse e maior investimento nas últimas duas décadas. Implementado inicialmente na prática pela Amazon, e anos mais tarde seguido por gigantes do mundo tecnológico como a Google, Microsoft, IBM, entre outros, a computação cloud permite fornecer poder computacional, armazenamento e largura de banda como um serviço. Teve como motivação principal a eficiência do processamento da informação que é obtida, devido à economia de escala, em grandes centros de computação. E teve sucesso entre as entidades individuais e empresariais por poderem beneficiar de uma infinidade aparente de recursos computacionais a um preço utilitário.

Com a previsão do crescimento da utilização de dispositivos IoT (Ex.: sensores para *smart cities*, *smart homes*, dispositivos *wearables*) e o aumento da exploração de aplicações com consumo elevado de largura de banda (Ex.: Video on Demand (VOD), streaming de conteúdo vídeo, 4k TV), prevê-se que a rede fique demasiado congestionada devido à grande quantidade de dados transferidos entre os dispositivos *front-end* (Ex.: telemóveis, *tablets*, *desktops*, computadores portáteis, televisões, consolas de jogos) e os centros de dados cloud. Também a distância entre estes dispositivos e os centros de dados cloud impõe uma limitação na interatividade das aplicações (Ex.: realidade virtual e aumentada) devido à latência elevada entre os mesmos.

Os avanços tecnológicos recentes nos dispositivos periféricos (Ex.: *routers*, *gateways*, *switchs* e estações base) indicam uma possível mudança gradual de paradigma de computação como tentativa de resolver esses problemas. O paradigma de computação distribuída usando nós computacionais periféricos, denominado por computação edge (*edge*

computing), pretende solucionar os problemas relacionados com a latência e a transferência elevada de dados entre dispositivos *front-end* e centros de dados *cloud*. O que é conseguido usando nós computacionais periféricos para fazer a ligação entre os dispositivos *front-end* e os centros de dados cloud. Como consequência, os dispositivos *front-end* usados pelos utilizadores podem beneficiar de uma redução de latência devido à proximidade aos dispositivos periféricos. Permite também a filtragem e agregação dos dados nos dispositivos periféricos reduzindo a quantidade de dados transferidos para a cloud.

No entanto, os recursos reduzidos dos dispositivos periféricos complicam a migração/replicação de aplicações monolíticas, para esses dispositivos, por serem demasiado grandes e complexas. Como solução, pode ser usada a arquitetura de micro-serviços que permite desenvolver software com base em múltiplos micro-serviços, de pequena dimensão e com *interfaces* bem definidas, que comunicam entre si através de mensagens. Cada micro-serviço pode implementar uma funcionalidade especifica e isolada permitindo, assim, o desacoplamento aos outros micro-serviços. Para além dos benefícios no desenvolvimento de aplicações baseadas em micro-serviços (ex.: módulos bem definidos, modularidade e reutilização, e desenvolvimento distribuído), a utilização desta arquitetura tem benefícios, quer na *cloud*, quer na *edge* (ex.: *deployment* independente).

Entre as desvantagens do uso de micro-serviços estão as falhas e atrasos em cascata em micro-serviços dependentes, e a necessidade dos micro-serviços que comunicam muito entre si serem lançados perto uns dos outros.

A solução desenvolvida contribui para a resolução da complexidade da gestão de micro-serviços nos centros de dados *cloud* e nos dispositivos na periferia da rede.

1.2 Problema

Considerando a heterogeneidade e dinamismo dos componentes periféricos, a diversidade dos micro-serviços e das suas interligações que compõem as aplicações, e a necessidade de um mapeamento dinâmico e eficiente dos micro-serviços na infraestrutura, o resultado é um sistema complexo impossível de administrar manualmente. Em particular, o problema da migração/replicação automática de micro-serviços resulta num problema complexo ao nível da monitorização do sistema e sua gestão dinâmica.

O objetivo deste trabalho foca-se neste aspeto particular da gestão dinâmica de aplicações baseadas em micro-serviços no contexto de infraestruturas heterogéneas *cloud*/*edge*. Nomeadamente no problema da migração/replicação dinâmica de micro-serviços (individual ou de micro-serviços relacionados), dos centros de dados *cloud* para a periferia, entre componentes periféricos e da periferia da rede para a *cloud*, tal que tenha em consideração as características e carga dos nós disponíveis na infraestrutura, bem como o volume e local de acessos dos clientes.

Este trabalho pretende completar um trabalho anterior, contribuindo para evoluir o sistema de gestão centralizado, para um sistema de gestores distribuído hierarquicamente,

e para o desenvolvimento de algoritmos de migração e replicação de serviços baseado nas localizações dos serviços, dos nós e dos clientes.

O processo de gestão automática, com avaliação e decisão das ações de reconfiguração dinâmica, baseia-se num conjunto de requisitos e métricas que são disponibilizadas por um subsistema de monitorização. Alguns exemplos de métricas que podem ser consideradas para a gestão, mas que não foram todas utilizadas neste trabalho, são a latência dos pedidos aos micro-serviços e volume de dados associados, a taxa de ocupação da rede, o nível de distribuição de carga pelos componentes periféricos e o custo monetário associado à computação de informação e à comunicação entre dispositivos. Estes valores podem assim ser usados para se tomarem decisões dinâmicas quanto à migração/replicação de micro-serviços e dos nós da infraestrutura considerando, entre outros fatores, os recursos dos componentes periféricos, a sua distribuição geográfica, o volume de dados na rede e a dependência entre serviços.

Na figura 1.1 é apresentada a visão simplificada da arquitetura do trabalho prévio. Onde é possível verificar que existe um gestor central, denominado "μServices Manag.", a executar na *cloud*, a controlar um conjunto de máquinas virtuais na *cloud* e de máquinas na periferia da rede, bem como um conjunto de contentores de serviços de aplicações, como por exemplo o e-μS B #1. É também possível ver dois componentes de apoio às operações do sistema, o balanceador de carga ("*Load balancer*"), que distribui os pedidos a serviços do tipo *frontend*, e o servidor de registo e descoberta de serviços ("*Service Registry*"), que permite registar e descobrir as réplicas registadas no sistema.

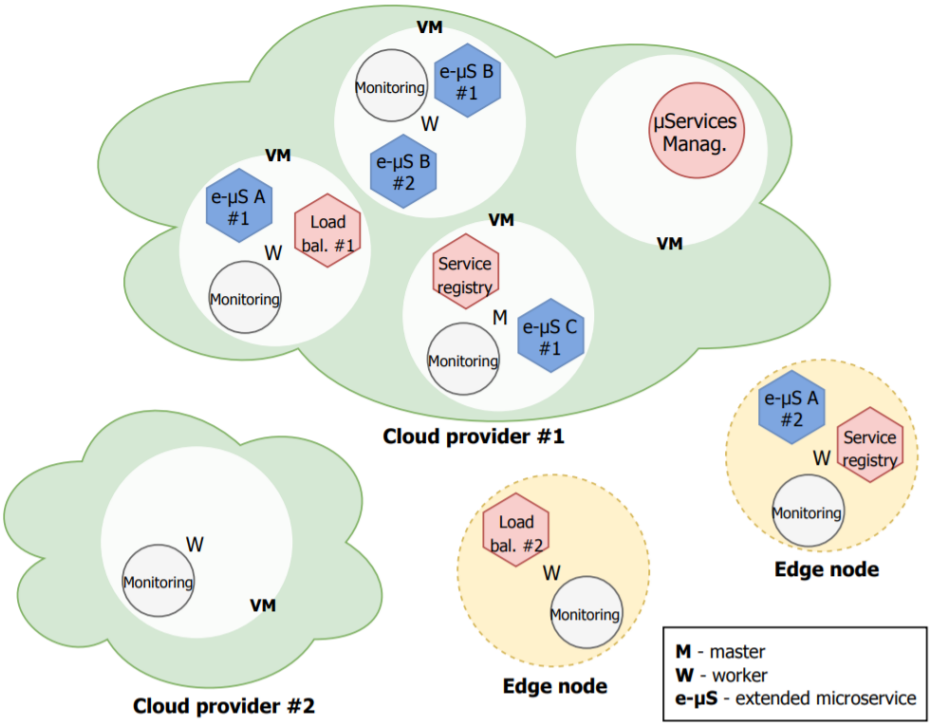


Figura 1.1: Arquitetura do trabalho prévio. Cortesia de [5].

Foram considerados alguns aspetos da arquitetura do trabalho prévio que podem constituir um problema para a eficiência do sistema de gestão, e que são os pontos de partida na definição da extensão da arquitetura e funcionalidades do trabalho anterior.

- **Limitação a uma única região**. A arquitetura do sistema anterior considera apenas uma região, sendo que existe apenas um componente de balanceamento de carga, e outro para registo e descoberta de serviços. Um sistema que abranja uma área geográfica grande, deve ter em consideração várias regiões, para conseguir distribuir a carga por diferentes instâncias de componentes. A arquitetura considerada tem em conta várias regiões, permitindo fazer uma gestão mais localizada por parte de um gestor local.

- **Componente de gestão centralizado**. O componente de gestão, responsável pela decisão da migração/replicação de micro-serviços ("μServices Manag."na figura 1.1), está centralizado num centro de dados *cloud*. Tal pode afetar negativamente a eficiência da solução anterior, devido à latência causada pelo distanciamento geográfico entre os nós do sistema, e ao número de contentores e nós que pode aumentar para números incomportáveis. No sistema atual são considerados mecanismos de gestão hierárquica do componente de gestão. Onde o gestor principal tem o conhecimento de todos os nós, contentores e eventos de ações de gestão (replicação/migração de contentores e inicio/término de nós) de todos os gestores, e os gestores locais recebem a informação das máquinas existentes, regras de gestão, e todos os dados necessários ao funcionamento do gestor local. Para além disso, o gestor local foca-se em fazer a gestão de uma região em concreto, onde o gestor principal tem a visão de todo o sistema.

- **Métricas de decisão**. No trabalho anterior, as métricas para a decisão das ações de gestão sobre os nós e serviços são limitadas à percentagem de Central Process Unit (CPU), percentagem de Random Access Memory (RAM) e número de *bytes* transferidos. Neste trabalho foi considerado que métricas adicionais devem ser tidas em conta no processo de decisão, de modo a melhorar a eficiência dos recursos e garantir a QoS das aplicações. Para além das métricas já usadas, a métrica do espaço do disco dos dispositivos periféricos permite evitar que seja tentada a transferência de imagens *docker* para uma máquina sem espaço disponível no disco. Outra métrica a considerar é a localização física dos utilizadores e dos contentores, cuja distância deve ser calculada, para tentar aproximar os utilizadores aos serviços.

- **Algoritmos de decisão**. No trabalho anterior, os algoritmos de decisão para onde migrar ou replicar os contentores são baseados em comparações entre palavras-chave de cidades, países ou continentes. Um dos problemas que surge, é a comparação entre palavras-chave em linguagens diferentes, ou entre diminutivos. Por exemplo, o algoritmo falha em detetar que ”lisbon” e ”lisboa” referem-se à mesma cidade.

Ou que "pt" e "portugal" referem-se ao mesmo pais. Em vez de comparações entre palavras-chave, a implementação de algoritmos baseados em distância, com o uso da latitude e longitude, permite obter resultados mais precisos e corretos.

- **Regras de gestão**. No trabalho anterior, as regras são aplicadas a *hosts* ou serviços. Mas existe a possibilidade de poderem ser definidas regras ao nível das aplicações, e individualmente, ao nível de cada contentor. A extensão à arquitetura tem esso aspeto em consideração, podendo ser possível a associação de regras a aplicações e a contentores.

- **Dependência entre micro-serviços**. Certos micro-serviços têm dependências a base de dados, como mySQL [1], MongoDB [2], ou Redis [3], e a certos serviços que devem executar no próprio nó para serem eficazes, como o memcached [4]. A gestão dos recursos na periferia tem em conta essas dependências, por forma a colocar esses tipos de serviços no mesmo nó.

- **Operações síncronas**. O sistema anterior usa operações síncronas, mesmo onde o uso de operações assíncronas e/ou paralelas poderia beneficiar a velocidade de execução do sistema. Com o suporte a diferentes regiões, essa situação é ainda mais agravada no gestor principal. Os componentes desenvolvidos devem ter em consideração o uso de operações assíncronas e/ou paralelas na obtenção de recursos externos ao sistema, como as instâncias virtuais a executar na Amazon Web Services (AWS) e alocação de endereços Internet Protocol (IP) fixos requisitados à AWS, obtenção das métricas de cada nó e contentor, na obtenção dos dados relativos aos acessos ao serviços, no lançamento e paragem simultânea de vários contentores e na iniciação simultânea de múltiplos componentes do sistema em mais do que uma região.

- **Balanceador de carga**. No trabalho prévio, havia apenas um tipo de aplicação, denominada *Sock shop* [5], e portanto o balanceador de carga apenas suportava essa mesma aplicação. Para que o sistema suporte várias aplicações, o balanceador de carga teve que ser modificado para que a sua configuração suporte mais do que um tipo de aplicação.

- **Interação e visualização do progresso do sistema**. No trabalho prévio, foi usada uma aplicação *web* para a visualização e interação do sistema, sendo que é preciso alterá-la, para incluir os aspetos desenvolvidos neste trabalho. Foi desenvolvida uma aplicação *web* completamente diferente em termos visuais, incluindo as funcionalidades do trabalho anterior juntamente com as funcionalidades novas do sistema

[1] mySQL: https://www.mysql.com/
[2] MongoDB: https://www.mongodb.com/
[3] Redis: https://redis.io/
[4] Memcached: https://memcached.org/
[5] Sock shop: https://microservices-demo.github.io/

atual, que permitir ao utilizador interagir manualmente com o sistema e visualizar o seu progresso em qualquer fase da sua execução.

1.3 Contribuições

A contribuição principal deste trabalho é a extensão de um sistema simplificado, já existente, de gestão e coordenação automático para migração/replicação de micro-serviços, entre componentes na periferia e centros de dados cloud. O objetivo da extensão ao sistema é torná-lo mais dinâmico, adaptando-se melhor às condições da infraestrutura e da utilização dos serviços, melhorar as decisões de gestão ao considerar os aspetos da arquitetura referidos na secção anterior 1.2, e suportar múltiplas regiões de execução. O desenho da arquitetura e o desenvolvimento inicial do sistema foi efetuado numa dissertação anterior [5]. A extensão do sistema contempla a:

- **Extensão e modificação da arquitetura e *middleware***. Mudança da arquitetura para suportar um modelo hierárquico de gestores que execute em várias regiões diferentes, com a extensão do middleware ao adicionar novas funcionalidades, nomeadamente:
  a) adaptação dos componentes do sistema que usam os valores das cidades/países/-continentes nos seus algoritmos de seleção de réplicas, para passarem a usar os valores das coordenadas;
  b) migração e replicação de contentores com base num algoritmo de seleção de localização baseado em coordenadas (latitude e longitude), na quantidade de pedidos dos clientes e na escolha do melhor nó do sistema para alojar uma réplica de um serviço, com base nos nós e nas máquinas na proximidade da coordenada calculada;
  c) extensão do balanceador de carga existente para suportar o registo e balanceamento de carga de serviços diferentes;
  d) iniciação de contentores e nós com proximidade a uma coordenada especificada, com base na escolha do melhor nó no caso do contentor, ou máquina no caso do nó, com base nas distâncias dos nós ou máquinas à coordenada selecionada;
  e) sincronização das replicas de serviços registados entre os servidores de registo a executar em diferentes regiões;
  f) implementação de gestores locais que gerem o seu próprio conjunto de contentores e nós;
  g) implementação de um sistema de sincronização de dados para manter os diferentes gestores do sistema sincronizados entre si;
  h) permitir a configuração de regras e métricas simuladas ao nível de aplicações e contentores, para além dos níveis já considerados atualmente: *hosts* e serviços;
  i) gestão do estado de instâncias virtuais na *cloud*. Com pedidos *on-demand* para iniciação, paragem, ou término de instâncias;

j) permitir a execução de comandos SSH e *bash* nos nós geridos pelo sistema, para *debug* ou interação manual com o sistema através da execução de comandos numa consola;
k) extensão à aplicação *web* para ser possível interagir manualmente e visualizar o comportamento de todos os aspetos do sistema, através de uma interface organizada e adaptativa aos acontecimentos do sistema;

E para a avaliação das implementações realizadas no sistema:

- **Avaliação**. A inclusão e adaptação de outras aplicações compostas por micro-serviços, para além da já usada (*Sock Shop*), nomeadamente, a *Hotel Reservation*, permite a verificação do desempenho e a correção das extensões realizadas ao sistema.

1.4 Organização do documento

Para além da **Introdução**, apresentada neste capitulo, o documento está dividido em mais cinco capítulos:

2. **Estado da Arte**. O capitulo 2 descreve as dimensões do estado da arte, nomeadamente, computação em cloud, computação na edge, arquitetura de micro-serviços, gestão, virtualização de recursos e trabalho relacionado.

3. **Solução proposta**. O capitulo 3 apresenta a solução proposta, incluindo a arquitetura proposta para o sistema hierárquico de gestão dinâmico de micro-serviços no continuo cloud/edge, com as suas funcionalidades, requisitos e limitações, extensões ao trabalho prévio, e todos os componentes necessários à sua execução.

4. **Implementação**. O capitulo 4 inclui a implementação do sistema, descrevendo as tecnologia utilizadas e os detalhes da implementação dos componentes estendidos e desenvolvidos.

5. **Demonstração, Validação e Avaliação experimental**. O capitulo 5 inclui uma demonstração da interface gráfica e descreve a validação e os testes executados para avaliar a correção e o desempenho do sistema.

6. **Conclusões e trabalho futuro**. O capitulo 6 conclui o trabalho efetuado nesta dissertação, incluindo sugestões para a sua extensão num trabalho futuro.

Capítulo 2

Estado da Arte

Este capitulo tem o objetivo de apresentar os conceitos e as definições necessárias para compreender o trabalho proposto no próximo capitulo 3, incluindo a secção 2.1 sobre a computação em cloud (*cloud computing*), na secção 2.2 é apresentada a computação na periferia da rede (*edge computing*), enquanto que na secção 2.3 é feita a comparação entre a computação na cloud e na edge. A secção 2.4 contém arquiteturas baseadas em micro-serviços. Na secção 2.5 são apresentadas as tecnologias de virtualização de *software*, dentro de contentores, como por exemplo, o *docker* [1], e as tecnologia de gestão de contentores, como o *docker swarm* [2] e o *kubernetes* [3]. Por fim, na secção 2.6 são estudados vários trabalhos relacionados com o trabalho apresentado nesta dissertação.

2.1 Computação em cloud

A National Institute of Standards and Technology (NIST) define a computação em cloud como sendo: *um modelo para permitir acesso ubíquo, conveniente e a pedido a um conjunto de recursos de computação partilhados (ex.: redes, servidores, armazenamento, aplicações e serviços) que possam ser rapidamente provisionados e libertados com um mínimo esforço de gestão ou interação do fornecedor do serviço [22].* A computação em cloud surgiu nos início do milénio, motivado pelo facto do processamento de informação em grandes sistemas de computação e armazenamento ser mais económico e facilmente acessível através da Internet. O acesso à computação pode ser visto como um serviço público semelhante à distribuição de água ou eletricidade [17]. Muitos fornecedores de serviços computacionais como a Google e Amazon (abordados na secção 2.1.3), IBM e Microsoft estão atualmente a promover este paradigma como uma utilidade [24].

[1] Docker: https://www.docker.com/
[2] Docker swarm: https://docs.docker.com/engine/swarm/
[3] Kubernetes: https://kubernetes.io/pt/

É um dos resultados do *Grid movement* [17] que tinha como objetivo desenvolver uma computação em grelha (*Grid computing*) constituído num sistema distribuído com um grande número de sistemas heterogéneos e diferentes domínios administrativos. O facto de serem utilizados sistemas heterogéneos resultou num problema de gestão difícil, com agendamento, alocação de recursos, distribuição de carga e tolerância a falhas [17]. Embora fosse um projeto popular na comunidade científica não teve impacto na indústria [17]. Aprendendo com as lições retiradas do movimento *Grid computing*, uma estrutura de suporte à computação em cloud é maioritariamente homogénea em termos de segurança, gestão de recursos e custos, tendo como alvo inicial a computação industrial [17]. Antes do aumento de popularidade da computação em cloud, os sistemas Peer-to-Peer (P2P) foram um dos grandes interesses da comunidade cientifica e industrial [17]. Os dois modelos têm diferenças relevantes, como o facto dos sistemas P2P serem auto-organizados e descentralizados, enquanto que os servidores na cloud têm um único domínio administrativo e uma gestão central.

De acordo com a entidade NIST, a computação em cloud tem cinco características essenciais, três diferentes modelos de serviço e quatro diferentes tipos [22]. As cinco características essenciais [22] são, nomeadamente, *self-service* a pedido, amplo acesso à rede, agrupamento de recursos, rápida elasticidade e medição de serviço:

a) ***Self-service* a pedido**. Esta característica vem do facto do consumidor conseguir adquirir automaticamente, e após necessidade, recursos computacionais como o tempo de computação, o armazenamento ou capacidade de rede sem ser necessária a interação humana com os fornecedores de serviço.
b) **Amplo acesso à rede**. As capacidades da cloud estão disponíveis através da rede e são acedidas por plataformas heterogéneas de clientes através de mecanismos padrão.
c) **Agrupamento de recursos**. Os diferentes recursos físicos e virtuais dos fornecedores cloud são agrupados para dinamicamente fornecer múltiplos consumidores dependendo da procura. Existe uma independência da localização porque normalmente o consumidor não tem controlo nem tem conhecimento da localização exata dos recursos a serem utilizados, mas pode ser possível a identificação da localização mais geral como o continente, pais ou centro de dados.
d) **Rápida elasticidade**. Os recursos devem ser elasticamente provisionados e libertados, em alguns casos automaticamente, de modo a ser possível escalar rapidamente dependendo da procura do consumidor. Desta forma, para o cliente é dada uma visão de que os recursos disponíveis são ilimitados e que podem ser consumidos em qualquer quantidade a qualquer momento.
e) **Medição de serviço**. Os sistemas cloud controlam e otimizam automaticamente a utilização dos recursos através da monitorização de capacidades métricas do sistema (e.g. armazenamento, processamento, utilização de rede, número de consumidores ativos).

Na computação em cloud é feita a distinção entre três modelos disponibilizados aos seus clientes, visualizados na figura 2.1. Estes modelos permitem isolar características

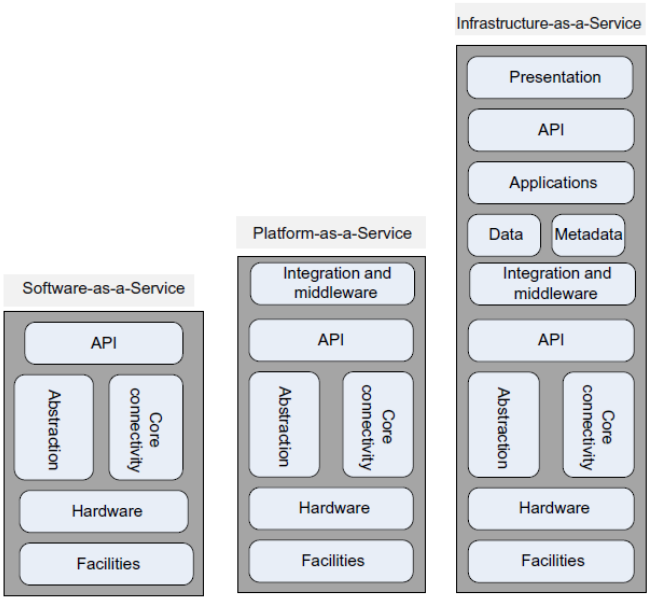


Figura 2.1: Diferentes modelos presentes na computação em cloud [17].

para lidar com as diferentes necessidades empresariais, educacionais e sociais [24].

- **Software as a Service (SaaS)**. O modelo SaaS fornece as capacidades necessárias para a utilização de aplicações disponibilizadas pelos fornecedores. O acesso a essas aplicações, por parte dos clientes, é feito através de uma interface bem definida. Neste modelo, o cliente não gere nem controla qualquer componente da infraestrutura cloud, apenas acede aos serviços através da interface a este disponibilizada. No modelo SaaS é comum o armazenamento dos dados ser efetuado noutro local relativamente longe da aplicação, não sendo, portanto, um modelo ideal para aplicações que não permitam o armazenamento externo dos dados. As características deste modelo, enumeradas anteriormente, também não são ideais para aplicações que requerem respostas em tempo real.
- **Platform as a Service (PaaS)**. O modelo PaaS é caracterizado por ter a capacidade de alojar aplicações criadas ou adquiridas pelos consumidores utilizando as ferramentas e linguagens de programação dos fornecedores cloud. Tal como no modelo SaaS, o cliente não gere os componentes da infraestrutura cloud, mas neste modelo existe controlo sobre as aplicações que desenvolve. Este modelo é particularmente bom na área de desenvolvimento de software para permitir colaboração entre vários utilizadores e, eventualmente, automatização do processo de desenvolvimento, lançamento e manutenção de aplicações. Aplicações que necessitem de ter um considerável nível de portabilidade, que utilizem linguagens de programação proprietárias, ou que necessitem do controlo da infraestrutura cloud, não são adequadas ao modelo PaaS.
- **Infrastructure as a Service (IaaS)**. O modelo IaaS fornece ao cliente um conjunto

de recursos presentes num sistema de computação, como, processamento, armazenamento e rede. Esses recursos a que o cliente tem acesso podem ser utilizados para executar o mais variado tipo de software. Tal como no modelo SaaS e PaaS, o cliente não gere os componentes da infraestrutura cloud, mas diferentemente dos outros dois modelos, o cliente tem controlo sobre o software que executa no sistema e que pode incluir sistemas operativos e/ou aplicações. Este modelo é utilizado principalmente para arrendamento de poder de computação utilizado para os mais diversos serviços.

2.1.1 Benefícios da computação em cloud

A computação em cloud tem várias vantagens por ser um modelo realista, utilizar tecnologias avançadas e recentes, ser conveniente para o utilizador e ser económico:

- **Modelo realista**. O modelo da computação em cloud é realista por ser homogéneo, tanto em hardware como em software, e ter um único domínio administrativo. Este foi um dos aspetos revistos e melhorados após o fracasso do, anteriormente referido, *Grid movement*. O facto de haver um sistema homogéneo e com um único domínio administrativo permite simplificar algumas soluções de problemas relacionados com sistemas de computação, como tolerância a falhas, qualidade de serviço, gestão de recursos ou segurança. Por exemplo, a criação de um conjunto de máquinas virtuais iguais e/ou similares, facilita o *deployment* das aplicações bem como as ações de gestão como atualizações, ou substituição em caso de falhas, nessas máquinas virtuais.
- **Tecnologias avançadas e recentes**. A computação em cloud é desenvolvida e promovida por grandes empresas e grupos comerciais, com grandes capacidades económicas, o que permite vastos investimentos em software, hardware, armazenamento, processadores e redes. Essa tecnologia é necessária para desenvolver novos e melhores sistemas de computação em cloud e competir por uma melhor posição de mercado [17]. O facto dessas grandes empresas terem a capacidade para alcançar uma distribuição global dos seus centros de dados permite também uma maior proximidade aos clientes nas diferentes regiões.
- **Escalabilidade**. A computação em cloud é altamente escalável e elástica por ser consistida por um conjunto homogéneo de sistemas de computação com uma política de pagamento baseada na sua utilização. As aplicações com elevada computação e utilização de dados que se possam particionar podem beneficiar de escalabilidade horizontal para melhorar os seus tempos de execução. E uma aplicação com crescente número de utilizadores pode tirar proveito da elasticidade da cloud de modo a suportar maior carga adicional com esforço mínimo dos *developers* da aplicação [17].
- **Paradigma cliente-servidor**. A computação em cloud baseia-se num paradigma

cliente-servidor e a maioria das aplicações cloud atualmente disponíveis tiram proveito de uma comunicação sem estado entre clientes e servidores. Numa comunicação sem estado, cada pedido é independente entre si e não é guardada informação entre pedidos do mesmo cliente. A utilização deste tipo de servidores tem vários benefícios como a sua rapidez e facilidade no estabelecimento de comunicações, facilidade de recuperação de falhas, bem como uma maior simplicidade, escalabilidade e robustez comparada com servidores com estado [17].

- **Preço utilitário**. Como a cloud utiliza uma política de pagamento baseada na utilização dos seus recursos, evita que os utilizadores tenham que investir numa infraestrutura e efetuar a manutenção de uma sistema de larga escala [17, 25]. Nos grandes centros de computação, devido à economia de escala, os fornecedores cloud conseguem um aproveitamento melhor dos recursos na cloud obtendo um baixo custo marginal de administração e operação [17, 25]. Devido ao baixo custo de operação é possível que os utilizadores utilizem os serviços cloud a preço reduzido comparativamente a outros centros de menor dimensão. O modelo de negócio praticado na cloud e o resultado da economia de escala permite tornar a computação em cloud cada vez mais popular. A popularidade é absolutamente crucial para que o negócio seja rentável visto que existe um enorme investimento em infraestruturas. Com um negócio rentável, é do interesse dos fornecedores investirem cada vez mais em centros de dados.

2.1.2 Limitações e desafios da computação em cloud

Apesar das diversas vantagens e desenvolvimentos tecnológicos proporcionados pela computação em cloud, ainda há obstáculos neste modelo que devem ser superados, como a garantia da disponibilidade de serviço, dependência ao fornecedor cloud, dependência da velocidade de transferência de dados, previsibilidade do desempenho, licenciamento de software, erros na infraestrutura cloud, segurança, monopólio do mercado, entre outros [17, 18]:

- **Garantia da disponibilidade de serviço**. Um desses obstáculos é garantir disponibilidade de serviço por parte dos fornecedores de cloud. Como a cloud pode ser partilhada por múltiplos utilizadores, o facto de existirem muitos utilizadores a usar o serviço simultaneamente não pode afetar negativamente a disponibilidade de serviço. Uma das soluções, embora não perfeita do ponto de vista económico, é garantir que existam recursos suficientes para satisfazer a previsão do maior número de utilizadores simultaneamente a usar o sistema.
- **Dependência ao fornecedor cloud** Outro problema associado com a computação em cloud está relacionado com a dependência dos utilizadores ao fornecedores cloud. Um utilizador ao usar um dos fornecedores cloud fica dependente do mesmo, sendo difícil a sua mudança para outro fornecedor. Uma solução atualmente em curso, mas não completa, passa por padronizar as tecnologias de modo a existir uma

menor dependência ao fornecedor cloud.

- **Velocidade da transferência de dados**. Certas aplicações na cloud utilizam muitos dados ficando bastante dependentes da velocidade de transferência de dados, o que pode causar um ponto de *bottleneck* no sistema. Uma das estratégias implementadas para aliviar o problema é armazenar a informação o mais perto possível do local onde é necessária. Quando a velocidade de transferência na rede é relativamente baixa, uma opção mais rápida e barata passa por armazenar os dados em memória não volátil e enviar os dados através de um método offline. À medida que a velocidade média de transferência dos dados aumenta ao longo dos anos, esta necessidade vai deixando de ser um problema.
- **Previsibilidade do desempenho**. Como dito anteriormente, a cloud é baseada na partilha de recursos para permitir uma política de pagamento utilitária. Por vezes, esta partilha de recursos pode ser problemática no aspeto da previsibilidade do desempenho esperado. Para permitir a elasticidade e escalabilidade da computação em cloud, são necessários novos algoritmos para o controlo de alocação de recursos e distribuição de carga capazes de escalar rapidamente.
- **Licenciamento de software** A atual tecnologia de gestão de licenciamento de software foi desenvolvida para ser utilizada em serviços não distribuídos. A computação em cloud fornece um serviço distribuído, não sendo a tecnologia de licenciamento de software atual adaptável às necessidades da infraestrutura que constituem a cloud.
- **Erros na infraestrutura cloud**. A infraestrutura cloud é complexa envolvendo múltiplos componentes o que torna a deteção de erros no sistema extremamente complexa e difícil. Outra razão para a dificuldade na deteção de erros deve-se ao facto do sistema poder envolver várias organizações com barreiras de segurança pouco definidas, um processo chamado *de-perimeterisation*. É também um problema para a determinação da responsabilidade de erros devido à cadeia complexa de eventos, em diferentes entidades, que muitas vezes é necessária para desencadear o erro. Com a responsabilidade do erro a ser partilhada por várias entidades, muitas vezes é difícil responsabilizar o conjunto de entidades pelo erro causado.
- **Segurança**. A computação em cloud reforça ainda mais alguns problemas relacionados com a ética computacional. Como o controlo da computação passa a ser delegado a um serviço de terceiros, existe maior risco potencial para situações de acesso não autorizado.

  A computação em cloud aumenta o armazenamento e a circulação de dados pessoais entre entidades o que agrava problemas relacionados com roubo de identidade devido ao acesso indevido dessa informação pessoal. Este facto pode colocar em causa o sucesso da computação em cloud devido ao aumento de desconfiança da sociedade perante a (in)segurança deste modelo. É do conhecimento da sociedade que os fornecedores cloud tenham armazenado uma enorme quantidade de sensíveis dados pessoais. A confiabilidade e auditabilidade dos dados é um importante

problema a superar.
Os recentes acontecimentos e acusações relacionados com a venda de dados pessoais por parte de empresas tecnológicas gigantes, caso do Facebook [9], impõe um grande obstáculo à aceitação da computação em cloud como um método viável e seguro para o armazenamento de dados pessoais. E, para complicar mais a situação, a privacidade é afetada por diferenças culturais. Há culturas que favorecem a partilha enquanto outras favorecem a privacidade. A computação em cloud pretende ser global e portanto devem ser discutidas soluções para este tipo de problemas culturais. É critico para o sucesso da computação em cloud que seja obtida a confiança da sociedade porque a cloud necessita de um grande número de utilização para que seja viável o grande investimento em infraestruturas feito pelas companhias fornecedoras de cloud.

- **Monopólio do mercado**. Apenas grandes companhias conseguem ter o poder económico para investir em infraestruturas cloud. Um outro problema está relacionado com o facto de apenas existirem algumas companhias que dominam o mercado. Existem preocupações sérias relacionadas com a manipulação de preços e políticas.

Com ainda importantes e difíceis problemas para serem superados, o futuro sucesso da computação em cloud está dependente da capacidade de promoção da computação utilitária, por parte das companhias e centros de investigação, para convencer uma maior população das vantagens da computação centrada em rede e conteúdo. É necessário encontrar soluções para aspetos críticos de disponibilidade e qualidade de serviço, escalabilidade e elasticidade, segurança e confiabilidade.

2.1.3 Casos de estudo

A Amazon foi a empresa pioneira que colocou em prática o conceito de computação em cloud à escala global. Após o estudo do potencial impacto que este tipo de computação poderia vir a ter na forma como temos acesso à computação, a aposta na computação em cloud foi reforçada por outros grupos empresariais, como a Google.

- **Amazon**. Os Serviços Web da Amazon, AWS, disponibilizam um conjunto de produtos baseados na cloud, incluindo computação (ex.: *Elastic Compute Cloud* [4]), armazenamento, bases de dados, analítica, capacidade de rede, móvel, IoT, ferramentas de desenvolvimento e gestão, e aplicações de segurança e empresarial. Com recurso a esses serviços, as organizações podem ser mais ágeis, diminuir os custos de tecnologia da informação e escalar mais facilmente [1]. Tem também desenvolvidas uma serie de soluções, baseadas nos seus próprios serviços AWS, para ajudar os *developers* a resolverem certos problemas comuns [5]. Dois dos serviços AWS, que se destacam por terem uma importância relevante no contexto deste trabalho, é

[4] https://aws.amazon.com/ec2
[5] https://aws.amazon.com/solutions

o *Amazon Elastic Container Service* [6] e o *Amazon Elastic Kubernetes Service* [7], que permitem o lançamento de contentores Docker [8] e a gestão de contentores usando Kubernetes [9], respetivamente, facilitando a execução e escalabilidade de aplicações conteinerizadas na plataforma AWS.

- **Google**. A Google Cloud Platform (GCP) é a plataforma de cloud da Google que fornece diversos serviços cloud para computação (ex.: *Google Compute Engine* [10]), armazenamento, rede, segurança, análise de dados, media, inteligência artificial e aprendizagem automática, IoT, gestão de Application programming interfaces (APIs), ferramentas para desenvolvimento e gestão de aplicações e migração de dados, de aplicações e de estruturas. Disponibiliza também ferramentas e produtos como o G-Suite, Google Maps, Cloud Identity, Chrome Enterprise, Android, Apigee, Firebase, etc, que usam os seus serviços GCP [12]. Em relação à tecnologia de contentores, a Google está diretamente associada à Kubernetes, por ter sido o *developer* inicial do sistema [15].
- **Microsoft**. Juntamente com a AWS e GCP, a Microsoft Azure [11] é uma das plataformas *cloud* com maior importância e quota de mercado [23], disponibilizando variados serviços [12] e produtos [13] *cloud*.

2.2 Computação na edge

A computação na edge é um novo paradigma promissor de computação em cloud descentralizado que coloca a aquisição de dados e funções de controlo, o armazenamento de conteúdo com grande utilização de rede, e as aplicações, mais perto do utilizador final. São utilizados dispositivos na periferia da rede (Internet ou rede privada) ligados a uma arquitetura de computação em cloud com maior capacidade de recursos [4]. É possível assim mover alguma carga computacional e aplicações dos centros de dados centralizados para a periferia da rede, de modo a aproveitar melhor os recursos computacionais atualmente pouco utilizados [27]. Deste modo, a computação na edge atua como camada intermédia entre o utilizador e os centros de dados cloud para oferecer um serviço com menor latência de rede [3]. Os dispositivos na periferia complementam as computações realizadas na cloud [27].

A origem da computação na edge pode ser associada ao aparecimento de Content Delivery Network (CDN) que utiliza nós na periferia para melhorar o desempenho da *web* ao fazer *cache* e *pre-fetch* de, principalmente, conteúdo com grande utilização de

[6] https://aws.amazon.com/ecs
[7] https://aws.amazon.com/eks
[8] https://www.docker.com
[9] https://kubernetes.io
[10] https://cloud.google.com/compute
[11] Microsoft Azure: https://azure.microsoft.com
[12] Serviços da Microsoft Azure: https://azure.microsoft.com/solutions
[13] Produtos da Microsoft Azure:

rede como é o caso do conteúdo multimédia. A computação na edge generaliza esse conceito ao ser integrado numa infraestrutura de computação em cloud distribuída. E em vez de apenas fazer *cache* de conteúdo web, o objetivo é ser possível executar código arbitrário nos nós periféricos [25]. Uma das possibilidades para permitir a execução de código arbitrário em dispositivos que normalmente estão adaptados a uma certa carga de trabalho é a utilização de virtualização, discutida na secção 2.5. O código a executar pode ser encapsulado em máquinas virtuais ou *containers* para existir isolação, segurança e gestão de recursos [25].

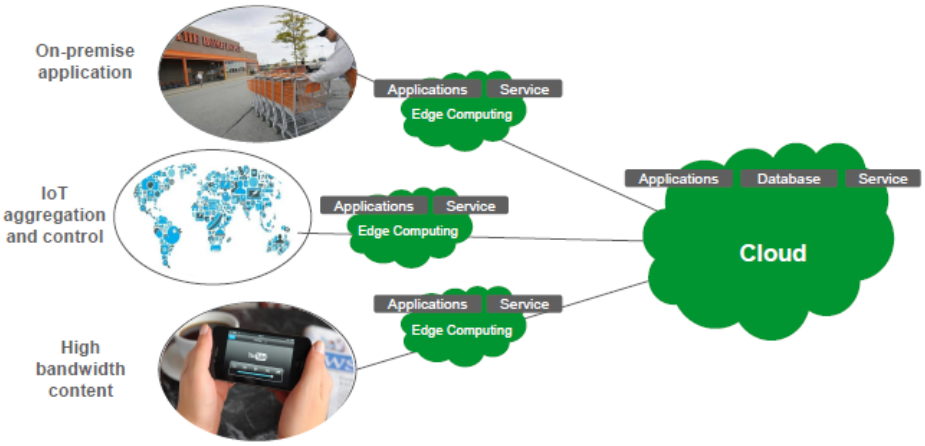


Figura 2.2: Representação da interação dos dispositivos na periferia com a computação na edge e cloud [4].

É possível distinguir na figura 2.2 três tipos de aplicações na periferia: aplicações on-premises, aplicações para agregação e controlo de dados IoT e distribuição de conteúdo com elevada utilização de rede:

- **Aplicações on-premises**. A garantia de disponibilidade é uma grande preocupação das aplicações on-premises (ex.: controlo industrial ou institucional). A utilização de componentes na periferia pode ajudar a superar desafios de disponibilidade no caso de, por exemplo, falha de energia prolongada ou um ataque Distributed Denial of Service (DDoS). As companhias que estão atualmente a utilizar a cloud no seu negócio, podem beneficiar da computação na edge para aumentar a disponibilidade do seu sistema ao introduzir redundância das suas aplicações criticas em componentes periféricos. E aumentar a segurança dos dados, visto que podem ser processados e/ou filtrados próximo onde são recolhidos [4].
- **Aplicações IoT**. As tecnologias associadas ao fenómeno de tornar tudo inteligente (ex.: cidades, agricultura, automóveis e saúde) requerem a utilização de grandes quantidades de sensores IoT [4]. As soluções atuais requerem a migração dos dados para centros de dados cloud para serem interpretados. O que implica uma latência grande, incomportável para aplicações de tempo real. E a grande quantidade de dados gerados por estes sensores IoT não é compatível com o paradigma da

computação em cloud porque a quantidade é de tal maneira elevada, que se torna incomportável transportar todos esses dados pela rede. As aplicações para agregação e controlo de dados IoT podem beneficiar da estrutura que compõem a computação na edge para reduzir a latência devido à maior proximidade aos componentes na periferia. E ao processar e/ou filtrar os dados gerados pelos sensores IoT nesses componentes periféricos, é possível reduzir a quantidade de dados transmitidos para os centros de dados cloud.

- **Aplicações com elevada utilização de rede**. As aplicações que requerem utilização elevada de rede (ex.: VOD, 4k Tv, video streaming) são as mais prováveis para congestionar a rede. Por forma a aliviar essa congestão, por parte dessas aplicações, os fornecedores do serviço estão a interligar um sistema de computadores com capacidade para disponibilizar mais rapidamente o conteúdo ao utilizador com menor congestão da rede. Esse sistema de computadores, que fazem *cache* do conteúdo, são um exemplo de computação na edge [4].

Avanços tecnológicos nos dispositivos perto dos utilizadores (ex.: dispositivos inteligentes, móveis e *wearables*, sensores, nano-centro de dados) e a previsão do aumento da sua utilização [14], podem ajudar a implementar essa mudança de paradigma devolvendo o controlo das aplicações, dados e serviços à periferia da Internet [11]. Esses avanços tecnológicos estão a permitir visionar novos tipos de aplicações aplicáveis em, por exemplo, cidades inteligentes, cuidados de saúde pervasivos, realidade aumentada, multimédia interativa, IoT e assistência cognitiva [3]. Mas, essas novas aplicações têm todas um problema em comum, são extremamente sensíveis à latência [3]. Outro problema está relacionado com a enorme quantidade de dados produzidos e usados por esse tipo de aplicações.

Um dos aspetos interessantes deste novo paradigma está relacionado com o facto da fronteira entre homem e máquina ficar menos definida. Ou seja, os humanos fazem parte da computação e das decisões relacionadas o que origina sistemas desenhados com foco nos humanos (*human-centered system design)*. Esta visão onde o aspeto principal é o papel fundamental dos humanos é referida por [11] como sendo a *Edge-centric Computing*. A computação P2P, conceito introduzido por volta dos anos 2000 com o aparecimento de sistemas de partilha de ficheiros, está bastante relacionado com a computação na edge. O paradigma *Edge-centric Computing* origina do P2P, mas em vez de tentar uma descentralização completa do sistema, expande o conceito de *peer* para todos os dispositivos na periferia da Internet e combina a computação P2P com a cloud.

Dois exemplos de *Edge-centric Computing* são a Mobile Edge Computing (MEC) e Mobile Cloud Computing (MCC) [24]. A MEC combina a operação de servidores periféricos com estações base para melhorar a eficiência da rede e a utilização de serviços considerando dispositivos móveis. A MCC pretende auxiliar na execução de aplicações nos dispositivos móveis usando nano-centros de dados (*cloudlets*) para deslocar a carga de tarefas dos dispositivos móveis para outros dispositivos com menos restrições de recursos

(energia, armazenamento, computação). Os *cloudlets* fazem a intermediação entre os dispositivos móveis e os servidores na cloud permitindo melhorar a Quality of Experience (QoE) dos utilizadores [24].

Outro tipo de computação na periferia é a computação fog. Enquanto que a computação na edge foca-se apenas na rede periférica, a computação fog procura usar tanto a rede na periferia como o núcleo da rede (ex.: routers principais, servidores regionais, wan switches). Este tipo de computação é particularmente bom para ser usado por dispositivos IoT porque os componentes fog podem ser colocados ao alcance desses dispositivos resultando numa significativa redução da latência. Ao contrário da computação na edge, a computação fog tem as capacidades para estender os serviços base da cloud (IaaS, PaaS, SaaS) [24]. Na figura 2.3, estão ilustrados todos os tópicos relevantes associados à computação na edge/fog. E na figura 2.4, estão visualizados os vários tipos de domínios de computação na edge anteriormente abordados.

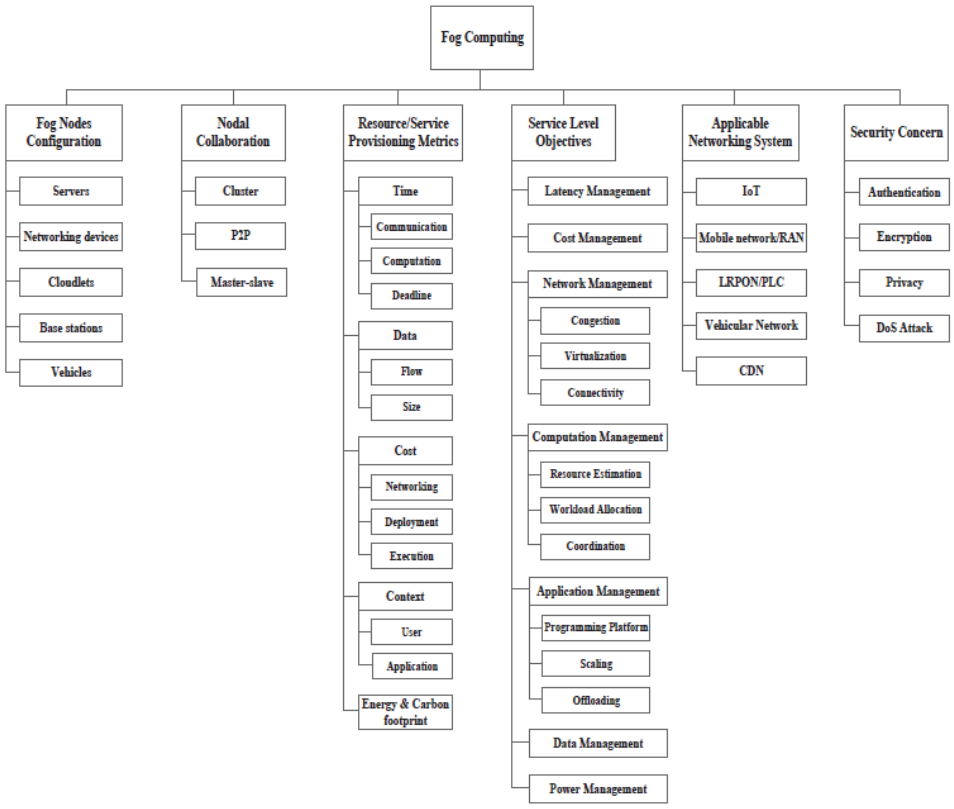


Figura 2.3: Taxonomia da computação na edge/fog [24].

Podem ser considerados vários dispositivos na periferia da Internet que podem ser usados para suportar a computação na edge como, por exemplo, *routers*, *gateways*, *switchs*, e estações base [3]. E componentes mais especializados como os dispositivos locais e centros de dados localizados e regionais [4]. Na figura 2.5 é possível a visualização da hierarquia que compõe a computação na edge/fog. Os dispositivos edge/fog permitem estabelecer um ponto intermédio entre os dispositivos dos utilizadores e os centros de

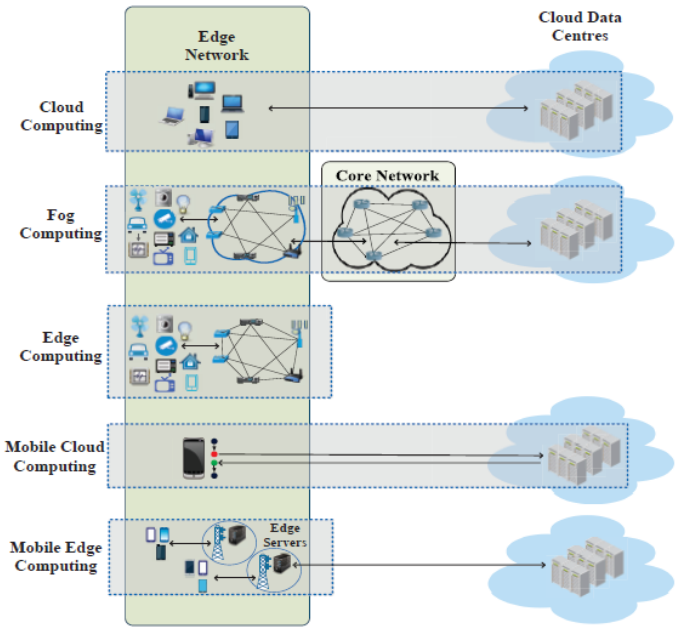


Figura 2.4: Os vários domínios da computação: cloud, fog, edge, mobile cloud e mobile edge [24].

dados cloud.

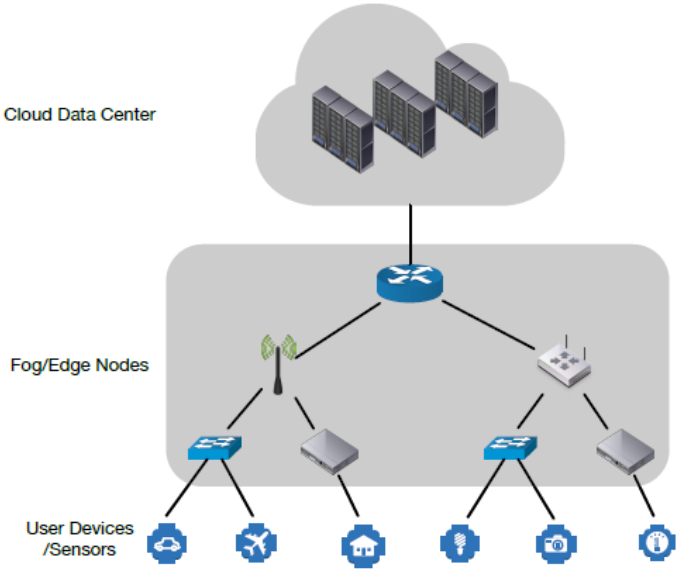


Figura 2.5: O modelo de computação na edge/fog [24].

2.2.1 Motivações e vantagens da computação na edge

Podem ser identificados alguns problemas importantes na computação em cloud que a computação na edge pretende solucionar, ou pelo menos, diminuir o seu impacto [11]:

- **Recuperação do controlo das aplicações alojadas na cloud**. A computação na edge pode devolver o controlo das aplicações e dos sistemas aos dispositivos periféricos. Com um desenvolvimento suficiente da segurança na computação na edge, poderá deixar de ser necessário que os utilizadores confiem unilateralmente numa entidade, como acontece no caso da computação em cloud com o fornecedor cloud.
- **Maior privacidade de dados pessoais e privados**. Atualmente, os dados pessoais são partilhados a serviços centralizados como sites e-commerce, serviços de classificação, motores de pesquisa, redes sociais e serviços de localização. A computação na edge introduz a oportunidade de filtrar, nos dispositivos periféricos, os dados pessoais e privados. Isto permite uma maior segurança e maior controlo dos dados enviados para os centros de dados cloud.
- **Oportunidade de utilização de recursos**. Com a utilização de um paradigma centralizado, está se a perder a oportunidade de exploração do grande potencial de computação, comunicação e poder de armazenamento presente nos dispositivos da periferia da rede. A computação na edge pode fazer um melhor aproveitamento desses recursos ao migrar parte da computação para a periferia.

As principais motivações para uma mudança de paradigma centralizado (computação em cloud) para um paradigma periférico (computação na edge) são a diminuição do tempo de transporte de dados (latência), a diminuição de transferência de dados e o aumento da disponibilidade [4, 3]. Esta mudança é particularmente importante sabendo que a utilização da Internet está cada vez mais associada a conteúdo com elevada utilização de rede. Esse tipo de aplicações (ex.: VOD, 4k Tv, video streaming) contribuem para a congestão na rede porque recolhem um elevado volume de dados que é tipicamente processado na cloud. A computação na edge coloca esse conteúdo e aplicações sensíveis à latência em dispositivos com poder computacional e com capacidades de armazenamento mais perto dos utilizadores [4].

A principal razão para o aparecimento da computação na edge é aproximar o nível de poder computacional atualmente presente em centros de dados centralizados aos utilizadores. Ao explorar novas aplicações na área da computação móvel e IoT, é óbvia a importância dessa aproximação porque afeta a latência e limita a largura de banda. Mesmo utilizando uma conexão direta com tecnologia fibra, a latência está limitada devido à velocidade da luz. E utilizar uma estratégia *multihop* para cobrir uma grande área usando muitos pontos de acesso limita também a latência e a largura de banda porque cada salto na rede aumenta o tempo de encaminhamento [25]. Este fenómeno pode ser observado na figura 2.6.

A proximidade dos nós periféricos, referidos por [25, 24, 3] como cloudlets, permite a existência de serviços cloud mais responsivo. Nova tecnologia, como a realidade virtual e aumentada, pode beneficiar da computação na edge ao processar a informação em cloudlets próximos sem afetar a interatividade e a experiência do utilizador. A proximidade dos cloudlets permite existir baixa latência, maior largura de banda e baixa instabilidade entre o utilizador e o dispositivo que faz a computação [25].

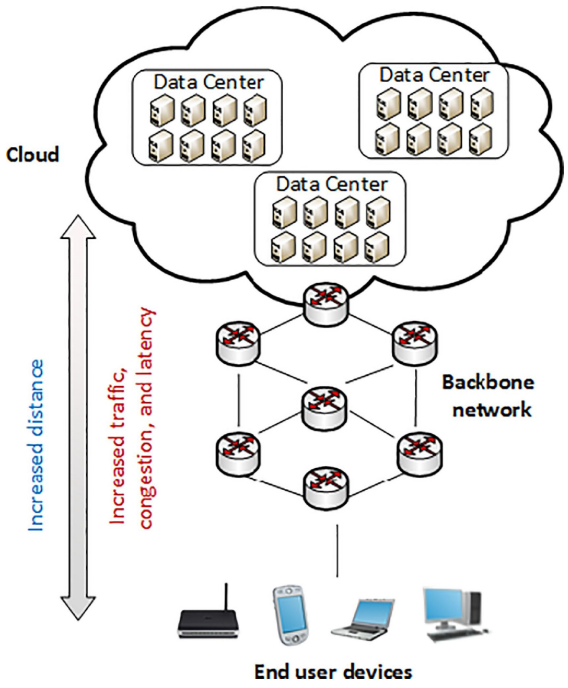


Figura 2.6: Problema da distância na computação em cloud [3].

Os dispositivos usados por utilizadores (ex.: computadores pessoais, telemóveis, tablets, dispositivos *wearables*) têm hardware relativamente limitado comparado com os servidores nos centros de dados. Os dados gerados por esses dispositivos normalmente têm que ser enviados para a cloud para serem processados. Mas, às vezes nem toda a informação enviada é necessária para construir o resultado enviado ao cliente. Existem dados que podem ser filtrados ou analisados em nós na periferia. Apenas a informação e os metadados extraídos após a analise precisam de ser enviados para a cloud aliviando a comunicação entre os dispositivos *front-end* e os centros de dados [27, 25].

A expansão rápida da utilização de serviços cloud tem como consequência inevitável o consumo crescente de energia por parte dos centros de dados cloud. Existe uma grande necessidade de adotar estratégias eficientes de consumo energético, não só devido a preocupações monetárias, mas também ambientais. A computação na edge permite incorporar uma gestão sensível de consumo de energia ao executar parte, ou toda, da computação necessária às aplicações e utilizadores, mais perto onde os resultados são usados [27]. O facto de ser reduzido o tráfego na rede a longa distância entre os utilizadores e os centros de dados cloud permite a existência de uma maior eficácia energética [3].

Funcionando como primeiro ponto de contacto na infraestrutura de cloud distribuída, um cloudlet pode restringir os dados que são enviados para a cloud como forma de garantir a sua privacidade [25].

Se um serviço cloud ficar indisponível devido a falhas de rede, falhas técnicas na cloud ou ataques DDoS, a utilização de cloudlets permite a ação de serviços que escondam temporariamente essa falha [25].

2.2.2 Desafios e limitações da computação na edge

A computação na edge ainda é muito recente e as *frameworks* atualmente públicas, que facilitem o processo de desenvolvimento, ainda são muito limitadas. Esse tipo de *framework* deve satisfazer requerimentos como, por exemplo, o desenvolvimento de aplicações que processem pedidos em tempo real nos nós periféricos. A implementação de processamento de dados em tempo real na periferia da rede ainda é um campo de estudo em aberto. Outro requerimento é a facilitação do *deployment* de aplicações em nós periféricos. É preciso saber onde colocar a carga da aplicação, estudar políticas de conexão e saber que nós utilizar de modo a otimizar a utilização da computação na edge. Para desenvolver tal *framework*, é necessário considerar alguns desafios a nível do *hardware*, *middleware* e *software* [27]:

- **Heterogeneidade de componentes.** Muitos nós computacionais localizados na periferia da rede (ex.: pontos de acesso, estações base, *gateways*, pontos de agregação de tráfico) têm características diferentes e funções especificas para a carga de trabalho que suportam. O objetivo da computação na edge é suportar qualquer tipo de computação e muitos nós na periferia estão adaptados para suportar apenas um certo tipo de computações [27, 24]. Um dos desafios passa por estudar como utilizar esses recursos na periferia de modo a suportar computação mais genérica. Usar técnicas de virtualização, como é o exemplo dos *containers* (ex.: docker e kubernetes, abordados na section 2.5) são sérios candidatos para superar este desafio.
- **Descoberta de nós periféricos**. De modo a explorar a periferia da rede são necessários novos mecanismos de descoberta para encontrar os nós que possam ser utilizados numa arquitetura de cloud descentralizada. Os métodos terão que ser bastante rápidos na identificação da disponibilidade e capacidade de recursos na periferia sem aumentar a latência ou comprometer a experiência do utilizador [27, 24].
- **Particionamento de tarefas e carga**. Um outro desafio vem do facto das tarefas e da carga terem que ser particionadas pelos nós na periferia. O particionamento eficiente e automático das tarefas, sem necessidade de indicar explicitamente as capacidades individuais de cada nó, é um grande desafio da computação na edge. É necessário o desenvolvimento de novas ferramentas de agendamento que particionem as tarefas pelos nós disponíveis [27, 24].
- **Garantia de qualidade de serviço**. O quatro desafio está relacionado com a garantia da qualidade de serviço dos nós na periferia. Esses nós terão que assegurar uma execução confiável das cargas de trabalho inicialmente pretendidas. A carga adicional de trabalho proveniente de dispositivos *front-end*, de outros nós periféricos ou da cloud, não pode comprometer de forma alguma o desempenho desses nós na execução das suas próprias tarefas [27, 24]. Na computação fog, o Service Level Agreement (SLA)[14] é afetado por diversos fatores (ex.: custo de serviço, utilização

[14] Especificação clara dos serviços que o cliente pode esperar do fornecedor.

de energia, características da aplicação, fluxo de dados, estado de rede). Portanto, num ambiente fog é bastante difícil especificar as medidas do serviço e os correspondentes Service Level Objectives (SLOs)[15] [24].

- **Segurança**. A utilização pública e segura de nós na periferia é outro desafio da computação na edge. É necessária uma definição clara dos riscos associados à utilização desses nós periféricos, tanto para os seus donos como para os seus utilizadores. A tecnologia relacionada com *containers* é um potencial candidato para uma utilização de nós na periferia com sucesso. Mas, ainda tem características de segurança pouco robustas que devem ser mais desenvolvidas [27, 24].
  Devido à enorme quantidade de tipos de dados, escala física, frequência de comunicação e variedade de utilizadores, é difícil definir as considerações relativas à segurança na computação na edge. Uma variável importante a ter em conta será a definição de segurança imposta pelos donos dos dispositivos periféricos que pode variar consoante as suas próprias necessidades e disposições [26].
- **Desenvolvimento de *frameworks***. Como os nós na periferia não têm todos os mesmos recursos, é necessário desenhar uma plataforma para desenvolvimento de aplicações distribuídas que tenha políticas de distribuição de tarefas pelos dispositivos periféricos e ajude na visualização de dados.
- **Novas tecnologias e definições padrão.** O nível de padronização nas várias camadas (ex.: infraestrutura, identificação, comunicação, descoberta, gestão, semântica e segurança) da computação na edge e dispositivos IoT ainda está pouco definido [26]. É necessário haver um esforço conjunto das entidades reguladoras para desenvolver novos padrões robustos devido à heterogeneidade dos componentes periféricos. Um dos aspetos importantes a definir é a forma de comunicação entre os dispositivos *front-end* usados pelos utilizadores e os dispositivos IoT com os nós periféricos que fazem a ligação aos centros de dados centralizados. As tecnologias de comunicação têm tido bastante evolução recentemente procurando melhorar a velocidade e confiança dos vários métodos de transmissão. Um dos padrões que se destaca são as especificações da família 802.x definidos pelo Institute of Electrical and Electronics Engineers (IEEE) [26]. Os desenvolvimentos recentes pretendem abordar problemas relacionados com a comunicação na periferia da Internet. Na área de troca de mensagens, pode ser usado o habitual Hyper Text Transfer Protocol (HTTP)/Hyper Text Transfer Protocol Secure (HTTPS). Embora devem ser considerados outros métodos mais especializados, que têm as suas vantagens, como o Message Queuing Telemetry Transport (MQTT), o Constrained Application Protocol (CoAP), o Advanced Message Queuing Protocol (AMQP) e o Data Distribution Service (DDS) [26].

[15] Características mensuráveis específicas associadas a um SLA.

2.2.3 Gestão de recursos na edge

Por forma a possibilitar o sucesso da computação na edge, é necessário existir uma gestão eficiente dos recursos dos dispositivos edge, que têm uma capacidade computacional muito inferior e uma distribuição geográfica muito maior, comparativamente aos recursos nos centros de computação cloud. A figura 2.7 mostra os vários domínios relacionados com a gestão de recursos na edge.

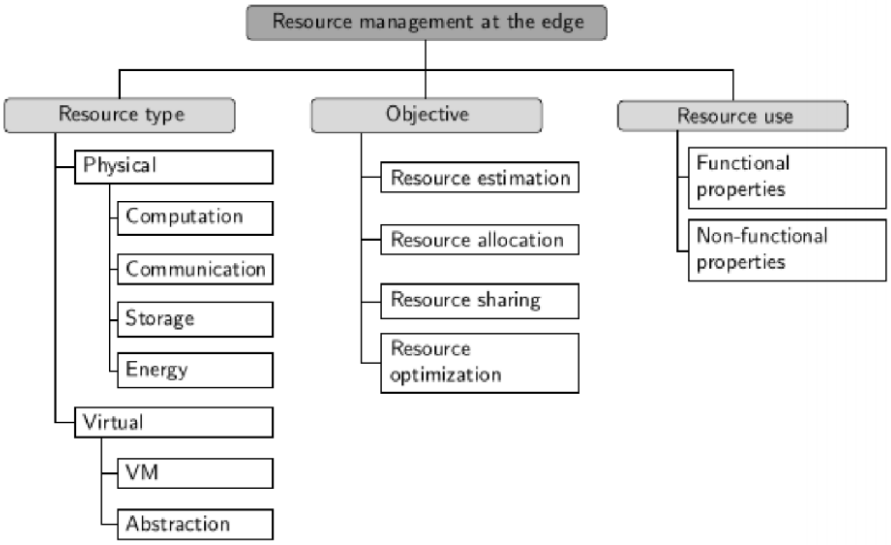


Figura 2.7: Os domínios da gestão de recursos na periferia da rede. [14].

A gestão de recursos considera duas categorias de recursos: físicos (*physical*) (ex.: capacidade computacional, como CPU, ou armazenamento) e virtuais (*virtual*) (ex.: máquinas virtuais, *containers*). E tem vários objetivos, como a capacidade de estimativa dos recursos (*resource estimation*) utilizados, a alocação dos recursos (*resource allocation*) para executar as tarefas, a partilha de recursos (*resource sharing*) como forma de otimização e/ou necessidade, e a otimização dos recursos (*resource optimization*), que está relacionado com os outros objetivos. O último aspeto da gestão de recursos aborda o propósito da utilização dos recursos, onde são consideradas as propriedades funcionais (*functional properties*), ou seja, o uso da sua capacidade de computação/armazenamento/etc, e as propriedades não funcionais (*Non-functional properties*), como o custo de migração de máquinas virtuais entre dispositivos.

2.3 Computação na cloud e edge

Na última década tem existido uma centralização e consolidação de serviços e aplicações em centros de dados originando o conceito de computação em cloud [11], que mudou radicalmente quase todos os aspetos da vida humana [3]. Os benefícios económicos da

computação em cloud, na secção 2.1.1, e o atual enorme investimento no seu desenvolvimento e recursos, realçam a sua importância na computação futura [25]. As aplicações baseadas na cloud utilizam, porém, servidores centralizados em centros de dados, que são acedidos por dispositivos na periferia. Devido aos recentes avanços tecnológicos e a novos tipos de aplicações, é insuportável continuar a enviar e receber toda a informação produzida e necessária para um centro de dados central. Com o aumento rápido do número desses dispositivos é colocada cada vez maior pressão nas estruturas computacionais cloud e na comunicação com os centros de dados. O que pode ter efeitos adversos na qualidade do serviço e na experiência do utilizador [27]. Também a preocupação crescente com problemas relacionados com a confiança, privacidade e autonomia associados à computação em cloud sugere claramente a necessidade de uma mudança de paradigma.

A computação osmótica (*Osmotic computing*) [28] é um paradigma recente, motivado pelo aumento da capacidade dos recursos na periferia da rede, e pelo desenvolvimento na área dos protocolos de transferência de dados, que facilitam a utilização desses recursos. Tem como inspiração o processo de osmose e evidencia a migração/replicação dos serviços (onde se encontrem em maior ”concentração”) para os vários locais (na cloud e na edge) onde são necessários mas escassos (menor ”concentração” de serviços). O objetivo da computação osmótica é permitir a existência um ambiente distribuído de *deployment* autónomo de micro-serviços, através da sua gestão dinâmica tanto em infraestrutura na cloud, como na edge, abordando problemas relacionados com o *deployment*, rede e segurança, e permitindo o suporte mais confiável de IoT.

2.4 Micro-serviços

O estilo de arquitetura de micro-serviços, baseado na computação orientada a serviços, pode ser definido como sendo a estruturação de uma aplicação distribuída como um conjunto de serviços coesos e independentes [20, 21, 6, 7]. Coeso no sentido de apenas implementar funcionalidades fortemente relacionadas com aquelas que pretende modelar [6]. E independente porque cada micro-serviço executa no seu próprio processo usando mecanismos leves para comunicação (ex.: pedidos de serviço web, ou Remote Procedure Call (RPC)) [20]. O facto dos micro-serviços serem independentemente *deployables* e escaláveis permite existir uma fronteira clara entre cada módulo das aplicações constituídas por micro-serviços, para além de permitir que o desenvolvimento de cada micro-serviço seja feito por equipas independentes [20].

A arquitetura de micro-serviços pode ser considerada como sendo o oposto do estilo monolítico, em que uma aplicação monolítica é construida como uma unidade única. Isto torna as aplicações monolíticas difíceis de usar em sistemas distribuídos sem uso especifico de *frameworks* ou soluções *ad hoc* (ex.: Network Objects, RMI, CORBA) [6]. É difícil de indicar uma definição formal de micro-serviços, mas este tipo de arquitetura tem várias características [20] que, embora não estejam presentes em todas as arquiteturas de

micro-serviços, podem caracterizar bem o tipo de aplicação, baseada em micro-serviços, esperada:

- **Componentização de software via serviços**. A Componentização de software pode ser descrita como sendo a construção de sistemas através da junção de vários componentes, sendo um componente uma unidade de software que é independentemente substituível e atualizável. Uma arquitetura de micro-serviços tem esta caraterística ao usar vários micro-serviços para compor um sistema.
  Outro tipo de componente são as bibliotecas, presentes em praticamente todas as linguagens de programação. Em comparação com as bibliotecas, que são usadas pelas aplicações monolíticas como chamadas de funções em memória, os micro-serviços têm a vantagem de serem independentemente "*deployable*". Ou seja, a mudança numa biblioteca da aplicação requerer um novo *deployment* de toda a aplicação. Enquanto que numa aplicação composta por múltiplos micro-serviços, a mudança num serviço apenas requer um novo *deployment* do próprio serviço, e potencialmente, de outros serviços dependentes.
  O estilo de arquitetura de micro-serviços disponibiliza apenas *guidelines* para a partição de componentes de uma aplicação distribuída em entidades independentes, não favorecendo qualquer tipo de paradigma ou linguagem de programação [6].
  Com a utilização de serviços como componentes há também uma melhor definição da interface exposta ao público, usando mecanismos de RPC, independentemente da linguagem de programação usada, o que permite evitar problemas relacionados com mecanismos de definição explicita de interfaces que cada linguagem de programação oferece. Contudo, a utilização destes mecanismos tem como grande desvantagem o facto de chamadas a procedimentos remotos terem maior custo, principalmente em termos de tempo e coordenação, do que chamadas a funções dentro do mesmo processo.
- **Organização baseada nas capacidades do negócio**. Normalmente a separação do desenvolvimento de uma aplicação grande implica a formação de várias equipas que se focam em partes diferentes da aplicação (ex.: equipa para interfaces ou equipa para bases de dados). Isto leva a que, a mais pequena mudança na aplicação possa ter que envolver a coordenação entre várias equipas.
  Usando uma arquitetura de micro-serviços, o desenvolvimento da aplicação pode ser separada em serviços tendo em conta as capacidades do negócio. Esta abordagem implica que as equipas sejam especializadas numa área, mas sim multidisciplinadas incluindo todas as áreas necessárias para a implementação e operação do serviço (ex.: experiência do utilizador, bases de dados, gestão de projetos). A arquitetura de micro-serviços tem a separação entre serviços explicita, tornando a fronteira entre equipas mais clara quando comparada com uma aplicação monolítica.
- **Produtos em vez de projetos**. O modelo de desenvolvimento de software normalmente utilizado tem o objetivo de completar um conjunto de funcionalidades, e

quando completo, é entregue a outra organização/equipa responsável pela manutenção do mesmo. A utilização da arquitetura de micro-serviços está associada a outra abordagem: a responsabilidade de todo o tempo de vida de um micro-serviço deve ser apenas de uma equipa, como se o serviço fosse um produto e não um projeto. Isto implica que os *developers* tenham que ter constantemente presente o modo de funcionamento dos seus serviços. Esta mentalidade também pode ser aplicada no desenvolvimento de aplicações monolíticas, mas a granularidade menor dos micro-serviços torna mais fácil a criação de relações pessoais entre os *developers* do serviço e os seus utilizadores.

- **Comunicação simples**. Muitos produtos, abordagens e estruturas (ex.: Enterprise Service Bus (ESB) com funcionalidades para *routing* de mensagens, coreografia, transformação, aplicação de regras de negócio, etc) enfatizam a utilização de mecanismos de comunicação complexas. A arquitetura de micro-serviços favorece outra alternativa: serviços coesos e comunicação simples. As aplicações desenvolvidas com micro-serviços normalmente recebem um pedido, aplicam a sua lógica apropriada e produzem um resultado. As comunicações podem ser simplesmente protocolos REST em vez de protocolos complexos de comunicação. Os dois protocolos mais utilizados são pedido-resposta HTTP, amplamente usado na web e com facilidade na realização de *cache* dos recursos, e a troca leve de mensagens, com implementações simples como o RabbitMQ e ZeroMQ.
  A forma de comunicação dos componentes numa aplicação monolítica é completamente diferente. Os componentes efetuam a comunicação através da invocação de métodos/funções em memória. A grande dificuldade em converter uma aplicação monolítica relativamente grande numa aplicação baseada em micro-serviços é a mudança do padrão de comunicação devido às diferenças de comunicação dos dois modelos.
- **Gestão de dados descentralizada**. O modelo conceitual do mundo é diferente entre sistemas. Por exemplo, a visão que um vendedor tem do cliente é diferente da visão de uma pessoa que dá suporte a clientes. Esta diferença está presente entre aplicações, mas também dentro da mesma aplicação, quando esta está dividida em vários componentes. Enquanto que as aplicações monolíticas preferem uma base de dados exclusiva para armazenar os dados persistentes, muitas empresas preferem a utilização de uma base de dados única para várias aplicações, para facilitarem a imposição das regras do negócio. Já na arquitetura de micro-serviços é preferível deixar cada serviço gerir a sua própria base de dados, por forma a manter a independência a outros serviços diferentes. A ilustração da diferença entre a gestão de dados pode ser vista na figura 2.8.
- **Automação da infraestrutura**. Muitos produtos e sistemas construidos com micro-serviços fazem uso extensivo de técnicas de automação de infraestrutura, que também podem ser usadas na construção, teste e operação de aplicações monolíticas, mas a forma de operação entre estes dois modelos é bastante diferente, como mostra

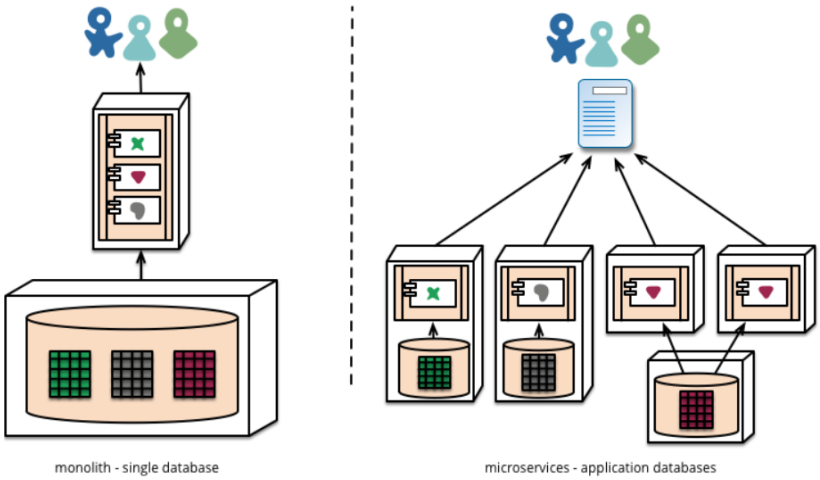


Figura 2.8: Comparação entre a gestão de dados em aplicações monolíticas e aplicações baseadas em micro-serviços. [20].

a Figure 2.9).

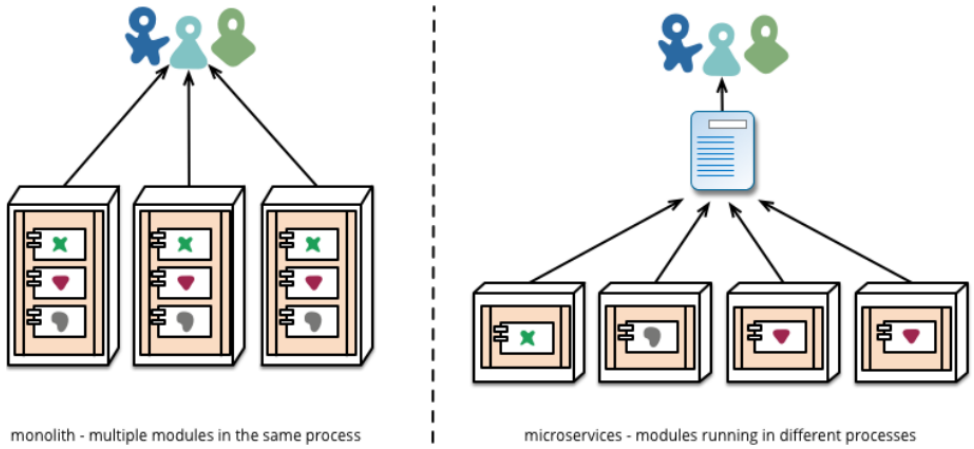


Figura 2.9: Comparação entre o modo de operação de aplicações monolíticas e aplicações baseadas em micro-serviços. [20].

- **Desenhado para falhas**. Uma das desvantagens da arquitetura de micro-serviços, comparativamente com o modelo monolítico, é a maior dificuldade da gestão de falhas. O desenho de aplicações baseadas em micro-serviços deve ter em consideração as falhas dos seus serviços. Como as falhas podem acontecer a qualquer momento, é importante existir uma capacidade de deteção rápida, e possivelmente, uma recuperação automática para garantir a disponibilidade do serviço. Consequentemente, o desenho de micro-serviços enfatiza muito a monitorização e o *logging*, em tempo real, do estado da aplicação, recolhendo dados cruciais (ex.: métricas de operação e negócio, estados, *throughput*, latência) para haja um aviso rápido de qualquer problema que possa existir no sistema. Este aspeto é particularmente importante na

arquitetura de micro-serviços, que implica a coreografia e colaboração entre serviços a executar em processos, e eventualmente, máquinas diferentes.

- **Desenho evolutivo**. O modo como são separados os componentes, como serviços, de uma aplicação é decisivo para o sucesso do desenho evolutivo numa aplicação baseada em micro-serviços. Os fatores principais para a formação de um componente são a substituição independente e capacidade de atualização, significando que um serviço deve ser substituído e/ou atualizado sem afetar demasiado os outros serviços da aplicação.

A utilização de micro-serviços pode ser vista como uma ferramenta que, através da agilidade da adição/remoção de serviços à aplicação, associando cada serviço a uma, ou várias funcionalidades, permite maior controlo das alterações na aplicação. Este facto é particularmente importante em aplicações que têm funcionalidades temporárias. Comparado com o modelo monolítico, a utilização de micro-serviços apenas é vantajosa se a aplicação for relativamente complexa, como mostra a figura 2.10. A utilização de micro-serviços em aplicações de menor dimensão apenas complicam a sua implementação. Por essa razão, muitas pessoas favorecem a utilização do modelo monolítico no inicio do desenvolvimento, mesmo sabendo que o uso da arquitetura de micro-serviços irá ser mais vantajosa no futuro. Mas essa abordagem tem outro tipo de complicações devido à dificuldade em construir uma aplicação monolítica, que seja facilmente transformada numa aplicação baseada em micro-serviços [21].

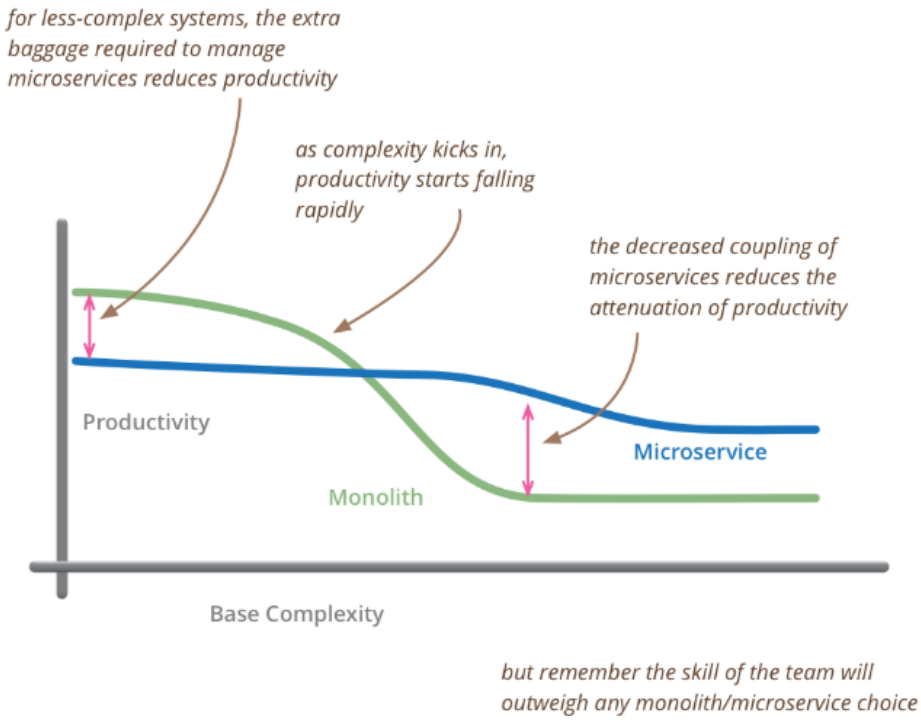


Figura 2.10: Comparação entre o beneficio de usar um modelo monolítico ou micro-serviços, considerando a complexidade da aplicação. [21].

2.4.1 Vantagens e desafios da arquitetura de micro-serviços

Tendo enunciado as características principais de uma aplicação baseada em micro-serviços, agora é mais simples perceber as vantagens e desafios da utilização desta arquitetura. Como a arquitetura de micro-serviços tem como base a computação orientada a serviços, esta herda os benefícios inerentes ao modelo orientado a serviços [6, 21]:

- **Fronteira entre módulos bem definida**. A arquitetura de micro-serviços tem como base uma estrutura modular, que é um aspeto muito importante para o sucesso de equipas de desenvolvimento grandes.
- **Dinamismo**. Um serviço, por ser independente e de relativamente pequena dimensão, pode ser facilmente replicado em novas instâncias para separar a carga no sistema.
- ***Deployment* independente**. O facto dos serviços serem independentes permite, quando os componentes da aplicação são devidamente separados, que exista um processo ágil para a remoção, adição e/ou alteração de serviços sem grandes dificuldades.
- **Modularidade e reutilização**. Sistemas complexos podem ser compostos por serviços mais simples. E o mesmo serviço pode ser utilizado em diferentes sistemas, partilhando o mesmo tipo de funcionalidade, promovendo reutilização.
- **Desenvolvimento distribuído**. Como os serviços são independentes entre si, cada equipa pode, após acordar na interface externa do respetivo serviço, desenvolve-lo em paralelo, em relação às restantes equipas.
- **Diversidade tecnológica**. A tecnologia usada em cada serviço é também independente. Logo, podem ser utilizadas linguagens de programação, *frameworks* e sistemas de bases de dados diferentes em cada serviço.
- **Integração de sistemas heterogéneos e legados**. Os serviços apenas necessitam de usar protocolos padrão para comunicar. Tudo o resto, desde o desenvolvimento, testes, ou manutenção, pode ser feito com qualquer tipo de linguagens/ferramentas/modelos, desde que adequadas para o efeito.
- **Características compatíveis com a computação na edge**. A arquitetura de micro-serviços tem características ideias para o seu uso em dispositivos com pouca capacidade computacional, como é o caso da computação na edge. A divisão de uma aplicação em componentes independentes tem, como consequência, uma maior eficiência da utilização dos recursos periféricos e uma rápida migração/replicação de serviços entre dispositivos e centros de dados cloud.

Mas a arquitetura de micro-serviços também tem alguns desafios [6, 21]:

- **Distribuição**. A arquitetura de micro-serviços é baseada num sistema distribuído, sendo a comunicação feita por RPC. Esta abordagem é mais difícil de programar, comparativamente com aplicações monolíticas, porque as chamadas de procedimentos remotos são mais lentas do que execuções de funções em memória e existe risco de falhas na comunicação entre serviços.

- **Consistência eventual**. O uso de consistência forte não é prática em sistemas distribuídos. Portanto, aplicações baseadas na arquitetura de micro-serviços têm de usar uma consistência mais fraca, como a consistência eventual.
- **Complexidade operacional**. As características do modelo de desenvolvimento de micro-serviços requerem uma equipa competente capazes de gerir os vários serviços, com *deployments* regulares, que compõem as aplicações.
- **Complexidade da rede**. A utilização de vários serviços independentes, com comunicação através de mensagens, pode resultar numa atividade de rede complexa. Esta complexidade dificulta a monitorização e deteção de erros da aplicação como um todo e pode expor o sistema a uma número maior de ameaças contra a aplicação.
- **Especificações do comportamento**. A definição de interfaces usadas para a comunicação entre os serviços não é o suficiente para garantir o funcionamento correto da aplicação. É necessário descrever o comportamento dos serviços, através de tipos de comportamento (*Behavioural types*), para garantir que os serviços tenham ações compatíveis entre si.
- **Maior possibilidade de ataques**. Enquanto que no modelo monolítico os ataques são restritos à máquina e ao sistema operativo, na arquitetura de micro-serviços, a interface externa dos serviços, que é exposta ao público através de APIs, é independente da máquina e até de linguagens de programação, ficando mais exposta a ataques.

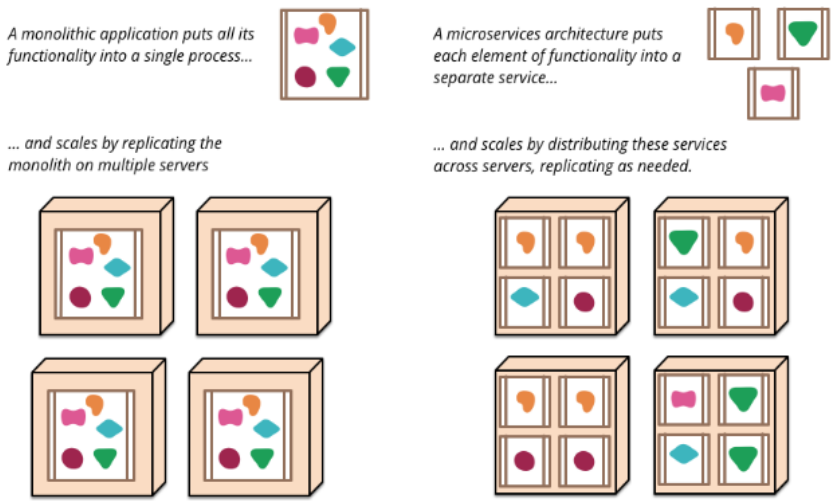


Figura 2.11: Comparação entre a escalabilidade de aplicações monolíticas com a escalabilidade de micro-serviços [20].

2.5 Virtualização

A complexidade da gestão de recursos de um sistema, discutida na secção 2.2.3, é quão grande quanto maior for o tamanho do sistema, a sua quantidade de utilizadores e diversidade de aplicações que usam esse sistema. Para além desses fatores externos, a gestão de recursos é afetada por fatores internos, como a heterogeneidade do hardware e software, redistribuição de carga e falha de componentes [19]. Uma solução para simplificar algumas tarefas de gestão de recursos, é designada virtualização de recursos, que permite que o estado de aplicações possa migrar ou replicar para outro servidor/dispositivo. Simplifica ainda a utilização de recursos proporcionando aos utilizadores um ambiente familiar para operarem, com isolação sobre os outros utilizadores [19]. A virtualização tem um papel fundamental na cloud:

- **Segurança**. Permite a isolação dos serviços que executam no mesmo hardware.
- **Desempenho e confiabilidade**. Permite que as aplicações migrem entre plataformas.
- **Desenvolvimento e gestão de serviços**. Permite o desenvolvimento e gestão de serviços por parte dos fornecedores cloud.
- **Isolação de Desempenho**. Os fatores externos não afetam tanto o desempenho de aplicações a executar em máquinas virtuais, permitindo a obtenção de métricas de desempenho mais previsíveis. É uma condição critica para garantir a QoS de aplicações a executar em ambientes de computação partilhados.

Existem duas tecnologias que permitem a virtualização de recursos:

a) **Hipervisor**. Através de máquinas virtuais (*Virtual Machine (VM)*), a virtualização é feita usando um ambiente isolado que parece estar a executar no seu próprio hardware, mas na realidade só tem acesso a um conjunto dos recursos. Com múltiplas VMs existe a ilusão de existirem várias instâncias do mesmo hardware, mas na verdade são todas suportadas por apenas um sistema físico [19].
b) **Containerização**. Através de *containers*, a containerização fornece uma camada adicional de abstração e automação de virtualização ao nível do sistema de operação. Permite a virtualização dos recursos com menos *overhead* quando comparado com o hipervisor porque enquanto que as máquinas virtuais apenas podem executar em cima de um sistema operativo, os *containers* não requerem o uso de nenhum sistema operativo [29]. Por esta razão, a escolha de *containers* para a virtualização de recursos em dispositivos periféricos parece ser a mais viável.

Com o foco na containerização como tecnologia de virtualização, podemos destacar duas soluções muito utilizadas: docker [16] e kubernetes [17].

[16] Docker: https://www.docker.com
[17] Kubernetes: https://kubernetes.io

2.6 Trabalho relacionado

A gestão de recursos e nos nós computacionais fog/edge é um desafio central para a computação na periferia, dada a pouca capacidade desses nós, a sua heterogeneidade e variabilidade [13]. Por exemplo, os nós podem falhar e a carga de trabalho/*workload* é variável no tempo, e.g. dependente do volume de acesso dos clientes, a partir de diferentes localizações geográficas, em que as aplicações competem pelos recursos escassos. A reserva dinâmica de nós computacionais com o *offload* e/ou replicação de computações/micro-serviços para a periferia é uma das formas que pretende dar resposta ao dinamismo do sistema. Tal requer sistemas de controlo/gestão, com arquitectura centralizada ou distribuída, *middleware* (e.g. com uma arquitectura hierárquica) disponibilizando serviços de monitorização de desempenho, comunicação, coordenação, etc. bem como algoritmos adequados. Os algoritmos compreendem dimensões como descoberta de recursos, avaliação de desempenho, distribuição da carga entre os nós existentes, ou decisão sobre a localização das computações, e.g. técnicas iterativas, ou decisões dinâmicas com base em condições e contexto do sistema [13].

Embora o paradigma de computação *serverless* recente permita a transferência de parte dessa gestão e configuração dos servidores (que suportam as computações) dos utilizadores para a plataforma *cloud*, a escolha entre o tipo de *deployment serverless* ou com base em micro-serviços pode depender do tipo de aplicaçoes suportadas. Os autores do artigo [8] fazem essa comparação, evidenciando, por exemplo, que as aplicações com base em micro-serviços podem não disponibilizar níveis de latência estáveis (como resultado do balanceamento da carga e redistribuição de tráfego) mas que o *deployment* e *caching* de micro-serviços é benéfico para pedidos repetitivos de pequena dimensão.

O trabalho proposto nesta dissertação apresenta um gestor distribuído hierárquico para o ciclo de vida e localização de nós computacionais e contentores de suporte aos serviços/componentes do sistema e de aplicações baseadas em micro-serviços. O gestor no topo da hierarquia e os gestores locais baseiam-se em decisões dinâmicas suportadas por regras e em pedidos *on-demand* por parte de utilizadores, e tendo em conta a localização geográfica dos pedidos e dos recursos. É ainda considerada, de uma forma simplificada, a dependência entre micro-serviços, podendo o sistema, reativamente, levar à criação de réplicas de serviços que são acedidos frequentemente por outros quando têm de responder a pedidos dos clientes (e.g. Frontend e catálogo de produtos).

Trabalhos recentes com soluções diversas para a gestão de recursos em computação *fog* são descritos em [16], cuja incidência é identificada como sendo, na sua maioria, nas dimensões de posicionamento dos recursos, escalonamento e *offloading*. Dada a complexidade do problema, obter soluções de balanceamento da carga com uma reserva óptima de recursos continua um problema em aberto, e são necessárias soluções descentralizadas que melhor se ajustem ao dinamismo e heterogeneidade dos nó *fog*. Os autores referem que a larga maioria dos trabalhos identificados se baseiam numa reserva estática de recursos em que a localização dos nós computacionais e dos componentes do sistema e

aplicações não varia durante o processo de gestão, e os recursos estão dedicados às aplicações avaliadas. Para mais, os algoritmos propostos são na sua maioria considerados num gestor centralizado, o que coloca problemas de escalabilidade dos recursos a gerir (e.g. no domínio IoT). O trabalho proposto nesta dissertação, por seu lado, pretende ser mais uma contribuição para um processo de gestão dinâmica de recursos que tem em conta a sua localização geográfica e a dos pedidos, e em que os própios gestores se encontram dispersos, mais próximos dos recursos a gerir.

Capítulo 3

Solução Proposta

O capitulo da solução proposta aborda, na secção 3.1 o trabalho feito numa dissertação anterior [5], na secção 3.2 é descrita a arquitetura proposta do sistema, a secção 3.3 descreve a arquitetura de um nó do sistema, na secção 3.4 é descrito o componente de gestão, a secção 3.5 contém os componentes de apoio às operações do sistema, na secção 3.6 é explicada a comunicação entre os componentes do sistema, e as funcionalidades e requisitos do novo sistema são abordados na secção 3.7.

3.1 Trabalho prévio

O protótipo anterior consiste num sistema simplificado com ênfase na execução transparente de micro-serviços na cloud e na edge. Esse protótipo, com arquitetura visível na figura 1.1, centra-se no contexto da gestão automática de aplicações de micro-serviços em ambientes cloud/edge, com o objetivo de replicar e migrar micro-serviços em nós da infraestrutura heterogénea. As decisões baseiam-se em métricas recolhidas aos nós e contentores, e em regras pré-definidas e associadas ao nível dos serviços e dos *hosts* do sistema.

O componente de gestão automática de micro-serviços permite que sejam definidas regras e métricas para utilizar na migração e replicação de contentores, por forma a melhorar a utilização e ocupação dos nós que executam os micro-serviços, e de aumentar e/ou diminuir o número de réplicas de contentores de serviços.

Um nó do sistema é um nó computacional (e.g. suportado por uma VM na *cloud*) onde é possível fazer o *deployment* de micro-serviços e cuja arquitectura é visível na figura 3.2. Inclui os componentes balanceador de carga, servidor de registo, monitor de pedidos de descoberta de serviços, *prometheus*, e eventuais contentores com serviços de aplicações. O

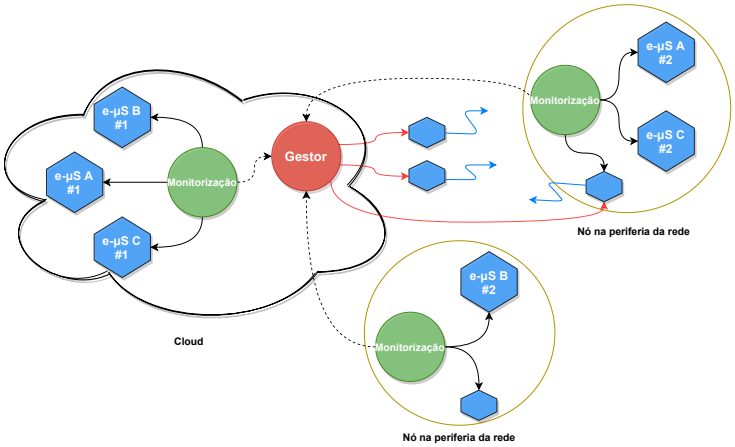


Figura 3.1: Arquitetura simplificada do trabalho prévio. Cortesia de [5].

balanceador de carga é baseado num servidor NGINX [1] e foi responsável pelo balanceamento de carga de acessos a serviços do tipo *frontend*, sendo apenas possível balancear a carga de réplicas de um único tipo de serviço. A distribuição da carga privilegia réplicas localizadas no mesmo pais, continente ou região, por essa ordem, do balanceador de carga. O servidor de registo tem a função de registar réplicas de micro-serviços, associando o nome do serviço a um endereço IP, possibilitando a descoberta de serviços por parte de outros serviços dependentes. O monitor de pedidos regista os pedidos de descoberta de micro-serviços a serviços dependentes, guardando os dados que depois são requisitados pelo gestor, para este decidir em que local deve colocar os contentores, baseado na localização dos pedidos. O *prometheus* e *node exporter* são componentes usados para obter métricas (Percentagem de CPU e percentagem de RAM) relativas aos nós geridos pelo sistema. As métricas dos contentores (*bytes consumidos e transferidos*) foram obtidas através do comando *docker stats* [2], disponível no *docker*.

A monitorização dos nós e dos contentores, a aplicação das regras, a tomada de decisão de replicação/migração/paragem dos nós/contentores, e a execução da decisão, é feita num processo de reconfiguração, denominado "*Control loop*", como apresentado na figura 3.3. Todos os passos do processo de reconfiguração são executados num intervalo de tempo configurável, que é 30 segundos por defeito.

O trabalho prévio incluiu ainda uma aplicação *web* para ver e gerir certos aspetos/parametrizações do sistema, como: aplicações, serviços, *edge hosts*, regiões, contentores, nós, servidores de registo, balanceadores de carga, regras e métricas simuladas.

Para a validação, foi usada a aplicação *Sock Shop* [3], onde foi demonstrada, através de *scripts* de testes k6 [4], a redução da latência no acesso a serviços a executar na periferia da rede, comparativamente com serviços a executar numa máquina na *cloud*.

[1] NGINX: https://www.nginx.com
[2] Docker stats: https://docs.docker.com/engine/reference/commandline/stats
[3] Aplicação Sock Shop: https://microservices-demo.github.io
[4] K6: https://k6.io/

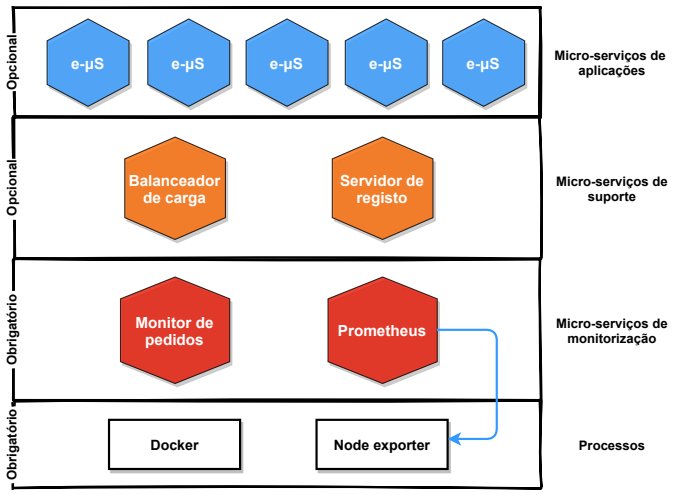
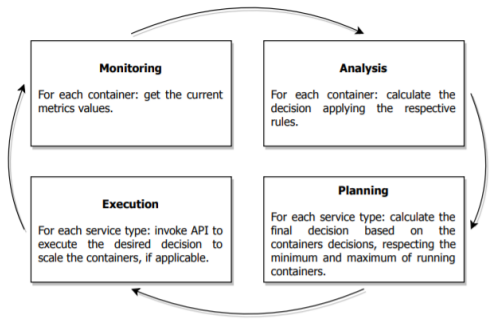
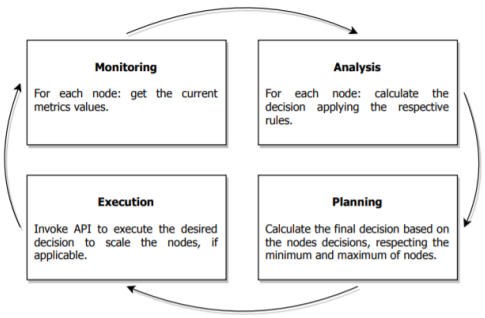


Figura 3.2: Arquitetura de um nó do sistema do trabalho prévio. Cortesia de [5].



(a) Parte do processo de reconfiguração dinâmica referente aos serviços.



(b) Parte do processo de reconfiguração dinâmica referente aos nós.

Figura 3.3: Processo de reconfiguração do sistema. Cortesia de [5].

3.2 Arquitetura do sistema de gestão dinâmico de micro-serviços na Cloud/Edge

A arquitetura proposta, na figura 1.1, contempla várias regiões, é, portanto, uma arquitetura distribuída hierárquica, onde em cada região pode executar um gestor local 3.4.2, um agente kafka 3.5.6, um balanceador de carga 3.5.5, e servidor de registo 3.5.2.

O gestor principal 3.4.1 executa numa das regiões, controlando todo o sistema. Para a sincronização dos dados entre os gestores, é usada uma solução com recurso ao Kafka 3.5.6.

3.3 Arquitetura de um nó do sistema

A arquitetura de um nó do sistema, figura 3.5, é semelhante à arquitetura do trabalho prévio, com a diferença da possibilidade de existirem componentes de agentes Kafka e gestores locais, a executar no nó. Os componentes obrigatórios (prometheus e monitor

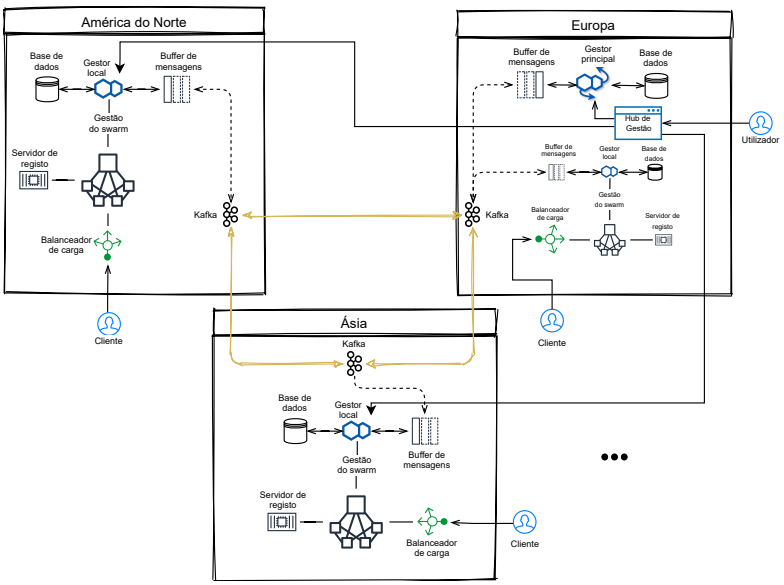


Figura 3.4: Arquitetura simplificada do sistema de gestão dinâmico.

de pedidos), juntamente com o processo Node exporter, são iniciados na altura em que o nó entra no *docker swarm*. Já o docker, é assumido que já está instalado e a executar na máquina assim que esta inicia.

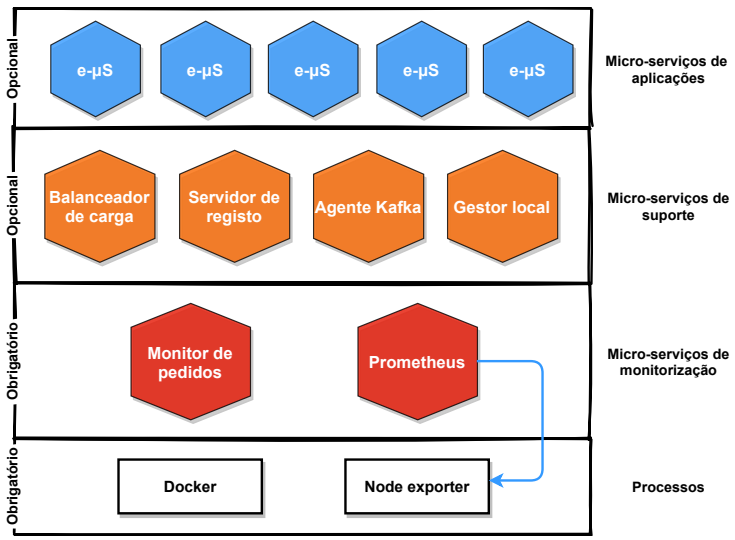


Figura 3.5: Arquitetura de um nó a executar no sistema.

3.4 Gestores

Os gestores são um conjunto de 3 componentes que, juntamente, formam a base do sistema de gestão implementado nesta dissertação.

3.4.1 Gestor principal

Baseado no gestor desenvolvido no trabalho prévio, o gestor principal é o componente que faz a gestão principal do sistema. As suas funcionalidades incluem:

- Agregar os dados de todos os gestores, aplicações, serviços, métricas simuladas e regras definidas.
- Disponibilizar um API para o Hub de gestão obter a informação do sistema, e poder gerir manualmente o mesmo.
- Alocar endereços IP elásticos através da AWS, para serem usados em situações onde o endereço deve ser conhecido à partida.
- Gerir os componentes de apoio às operações, incluindo gestores locais, servidores de registo, agentes kafka e balanceadores de carga.

3.4.2 Gestor local

O objetivo de um gestor local é gerir um conjunto de nós e um conjunto de contentores, numa determinada região, através da monitorização, analise de métricas, aplicação de regras, e execuções de ações. Um gestor local não comunica diretamente com o gestor principal. Em vez, a comunicação é feita para o agente Kafka que executa na mesma região, que por sua vez pode ser consumido pelo gestor principal quando estiver disponível.

As suas funcionalidades incluem:

- Monitorizar um conjunto de nós e contentores, obtendo métricas sobre os mesmos.
- Iniciar ciclos de adaptação, analisando as métricas obtidas, aplicando as regras definidas, e tomando decisões com base nisso.
- Disponibilizar um API para o Hub de gestão obter a informação do seu sub-sistema, e poder gerir manualmente o mesmo.
- *Heartbeats* enviados ao gestor principal, para este saber o estado do gestor local.
- Envio da monitorização e ações tomadas para o gestor principal.

3.4.3 Hub de gestão

O objetivo do Hub de gestão é permitir que um utilizador tenha a visualização do estado do sistema, e poder interagir manualmente com o mesmo. Permite modificar, adicionar e remover aplicações, serviços, regras e métricas simuladas. Permite dar ações aos gestores, por exemplo, lançar contentores e nós, controlar o estado dos contentores, nós e instâncias virtuais na *cloud*. Ou até lançar os componentes de apoio: balanceadores de carga, servidores de registo, gestores locais, e agentes Kafka.

Para além das funcionalidades, inclui características não funcionais, comuns às aplicações *web*: *feedback* de ações, listas paginadas, filtros, formulários, entre outras.

3.5 Componentes de apoio às operações

Os componentes de apoio às operações são um conjunto de componentes necessários para implementar as funcionalidades do sistema. Desde o registo e descoberta de serviços, à obtenção de métricas, balanceamento de carga do serviços, ou comunicação de dados entre gestores.

3.5.1 Proxy com autenticação básica

O Proxy com autenticação básica foi incluído no trabalho anterior para proteger a informação guarda pelo *docker*, em cada nó do sistema. Continua a estar presente na arquitetura do sistema.

3.5.2 Servidor e cliente de registo e descoberta de serviços

O servidor e cliente de registo de serviços foram incluídos na arquitetura do trabalho prévio para permitir a descoberta de serviços, por parte de outros micro-serviços. Para o servidor, foi usado o Eureka [5] da Netflix. Como agora o sistema considera várias regiões, com um servidor de registo presente em cada região, foi preciso implementar uma solução que permita que os servidores repliquem as réplicas dos micro-serviços registados. Para tal, foram usados endereços IP elásticos, alocados na AWS, o que permite associar endereços IP públicos conhecidos no inicio da execução do sistema, a instâncias *cloud*.

No cliente de registo, o algoritmo para a descoberta de serviços usa o valor da cidade, país e continente associado a cada réplica de micro-serviços. Devido a limitações na comparação de palavras, o algoritmo foi alterado para usar coordenadas (latitude e longitude). Usando um algoritmo baseado em coordenadas, permite calcular as distâncias às réplicas, e selecionar a réplica com base em distâncias. Para evitar que uma réplica seja selecionada por muitos serviços dependentes próximos, foram incluídos mais dois algoritmos que usam distâncias juntamente com aleatoriedade entre um conjunto de instâncias.

No trabalho anterior também foram desenvolvidas APIs para facilitar a comunicação entre um micro-serviço e o cliente de registo e descoberta de serviços. Apenas foram considerados APIs para micro-serviços implementados em Java e Go. Agora, com micro-serviços implementados em Java, Go, Python, C, C++ e NodeJS incluidos no sistema, foram adicionados mais APIs para suportar as linguagens adicionais.

[5] Eureka Netflix: https://github.com/Netflix/eureka

3.5.2.1 Limitações

Como o endereço IP público de um servidor de registo deve ser conhecido à partida, este só pode ser iniciado na *cloud* e nunca numa máquina *edge*.

3.5.3 Monitorização de pedidos a serviços externos

O monitor de pedidos a serviços externos foi usado no trabalho prévio para permitir associar a localização e o número de acessos de serviços, por parte de micro-serviços dependentes.

Este componente foi modificado para, em vez de monitorizar as cidades/países/continentes associadas a cada réplica de serviços, passar a usar os valores das coordenadas (latitude, longitude).

Ao recolher a informação sobre os pedidos dos serviços, um gestor pode aceder aos seus dados para basear a escolha da melhor localização para onde migrar ou replicar um contentor de um serviço. Desta forma, o local escolhido tenta colocar a réplica do serviço o mais próximo possível dos seus serviços dependentes e clientes.

3.5.4 Prometheus e Node exporter

Para além das métricas simuladas incluídas nos gestores, e de modo a poderem ser recolhidas métricas reais sobre as máquinas geridas pelo sistema, no trabalho anterior foi incluído o Prometheus [6], juntamente com o Node exporter [7], que executa em todos os nós do sistema.

Este componente é para ser substituído por um outro componente desenvolvido noutra dissertação de mestrado, portanto, a monitorização não foi o foco deste trabalho.

Continua a ser utilizado da mesma forma como era utilizado na arquitetura do trabalho anterior, para obter métricas dos nós geridos pelos gestores.

3.5.5 Balanceador de carga e API de configuração

No trabalho anterior, o balanceador de carga foi baseado num servidor NGINX, usado para distribuir os pedidos a serviços *frontend*, mas a sua configuração apenas suporta o registo e redirecionamento de um tipo de serviço *frontend*. Como agora o sistema contém serviços *frontend* de aplicações diferentes, idealmente deve suportar vários tipos de serviços *frontend* em simultâneo. Portanto, o *template* que gera o ficheiro de configuração NGINX foi alterado para suportar essa funcionalidade. O que implicou que o API para adicionar/remover/obter os servidores teve que ser alterado para suportar as novas operações.

O componente usava a versão 1 do Maxmind GeoIP para associar a localização dos utilizadores aos pedidos feitos, e com acesso a essa informação, conseguir escolher o melhor

[6] Prometheus: https://prometheus.io
[7] Node exporter: https://github.com/prometheus/node_exporter

local para replicar e migrar contentores. Como a versão 1 foi descontinuada e as bases de dados já não são atualizadas, foi modificado para usar a versão 2 do Maxmind GeoIP [8]. E como a versão 2 já não vem incluída com o servidor NGINX, teve que ser instalado como um módulo dinâmico a partir do repositório no GitHub ngx_http_geoip2_module.

O balanceador de carga foi também atualizado para deixar de usar a região na escolha do servidor, visto que, com um balanceador de carga por região, todos os servidores registados estão na sua região. Agora, o pedido é direcionado para o servidor com menos pedidos, dentro da sua região.

3.5.5.1 Limitações

Os balanceadores de carga desconhecem os servidores registados em balanceadores de carga noutras regiões, o que pode implicar que situações como ilustrado na figura 3.6 possam acontecer. Neste caso, o utilizador recebe uma resposta *404 Not found*, a indicar que nenhum servidor do serviço sock-shop-frontend foi encontrado, quando existia uma réplica do serviço a executar na Europa.

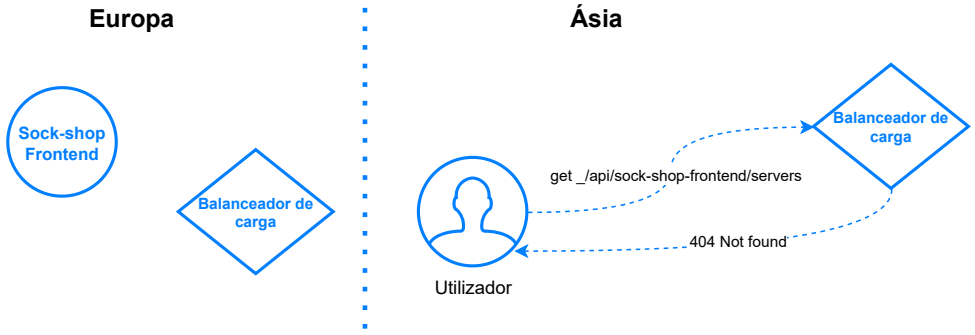


Figura 3.6: Limitações do balanceador de carga.

3.5.6 Agente Kafka

O Kafka é o método escolhido para ser feita a sincronização dos dados entre o Gestor principal e os Gestores locais, e vice-versa. Um agente Kafka é um nó do *cluster* gerido pelo sistema kafka, para onde podem ser replicados os registos dos tópicos. É lançado um agente kafka nas regiões onde existe um gestor local, para que este tenha acesso rápido aos dados transmitidos pelo gestor principal.

3.6 Comunicação entre componentes

No sistema são usados vários métodos de comunicação, visível na figura 3.7, quer entre os componentes, quer entre os micro-serviços. O método de comunicação escolhido depende das garantias e qualidades do método, bem como das necessidades de cada componente.

[8] Bases de dados Maxmind geoip2: https://www.maxmind.com/en/geoip2-databases

Na secção 3.6.0.1 é descrito o uso do Kafka para a comunicação entre o Gestor principal e os Gestores locais, na secção 3.6.0.2 o uso de HTTP para a comunicação entre certos componentes do sistema, como, por exemplo, o Hub de gestão e o Gestor principal, nas secções 3.6.0.3 e 3.6.0.4 o uso de RPC e RabbitMQ, respetivamente, usado na comunicação entre micro-serviços.

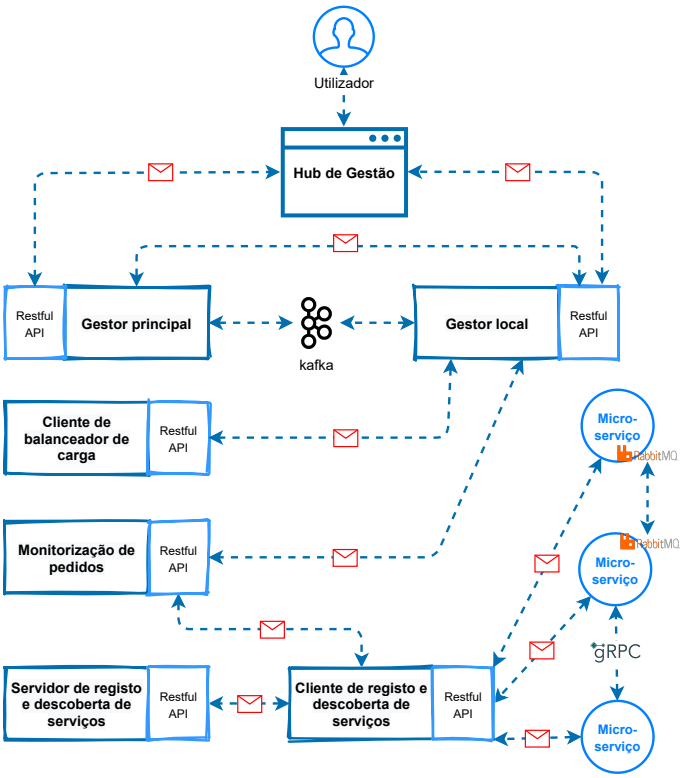


Figura 3.7: Comunicação entre os componentes do sistema.

3.6.0.1 Kafka

A sincronização dos dados entre o Gestor principal e os Gestores locais é feita através do Kafka [9], que é uma plataforma distribuída de *streaming* de eventos. Foi escolhido o Kafka como meio de comunicação para a transmissão dos dados porque no inicio da execução de um Gestor local, permite que este consuma a informação toda desde o inicio da execução do sistema, ficando com uma versão completa e atualizada dos dados. Outra vantagem do Kafka é o facto de que o Gestor principal não precisa de disseminar a informação diretamente para os Gestores locais, mas apenas para um dos agentes Kafka. O que permite que o sistema seja capaz de se expandir horizontalmente, através da adição de mais Gestores locais nas zonas necessárias. Os agentes são lançados dinamicamente em cada região, conforme haja a necessidade de existir um Gestor local nessa região.

[9] Kafka: https://kafka.apache.org

3.6.0.2 HTTP

O protocolo HTTP foi usado para a comunicação entre certos componentes do sistema, seguindo uma arquitetura Representational State Transfer (REST). Todos os componentes que pretendam receber pedidos HTTP, expõem uma API através de um servidor HTTP.

O cliente acede ao Hub de gestão para interagir manualmente com o sistema, sendo que a comunicação ao Gestor principal é feita através de uma API que expõe *endpoints* para obter, adicionar, alterar e apagar dados na base de dados, bem como suportar as funcionalidades do Hub de gestão. Em relação a operações que dizem respeito a um certo Gestor local, o Hub de gestão é capaz de direcionar o pedido diretamente para o gestor em questão, como é o exemplo de migrar/replicar contentores, ou efetuar alterações ao nível dos nós geridos pelo gestor.

O Gestor principal comunica com o Gestor local através de HTTP, para redirecionar certos pedidos de clientes: lançar aplicações e serviços, lançar e sincronizar contentores, adicionar, remover e sincronizar nós.

O gestor local usa o API disponibilizado pelo Cliente de balanceador de carga para atualizar os servidores registados dos serviços *front-end*, e o API do monitor de pedidos, para obter informação sobre os pedidos a serviços externos por parte dos micro-serviços, que é usada para a escolha do local de migração de contentores, como descrito no algoritmo 1.

O Cliente de registo acede ao API do Servidor de registo e descoberta de serviços para registar e obter as instâncias de serviços. E acede ao API do monitor de pedidos para registar pedidos a serviços externos por parte dos micro-serviços.

Por fim, o Cliente de registo define uma API que é acedida pelos micro-serviços para se registarem e obterem instâncias de serviços dependentes.

3.6.0.3 RPC

RPC é uma tecnologia de comunicação entre processos, que permite a execução remota de procedimentos. É usado por alguns micro-serviços para comunicarem entre si, como é o caso nas aplicações *Hotel Reservation* 5.3.0.2 e *Online Boutique* 5.3.0.5, onde os serviços comunicam através de chamadas RPC, usando a implementação da Google, gRPC [10].

3.6.0.4 RabbitMQ

Para comunicações mais complexas entre micro-serviços, é usado o RabbitMQ [11], que é um sistema de transmissão de mensagens, com suporte a transmissão assíncrona e *deployment* distribuído. É usado na aplicação *Sock shop* 5.3.0.1, entre os micro-serviços *Shipping* e *Queue-master*, e na aplicação *Social Network* 5.3.0.3, entre os micro-serviços *Compost Post* e *Write Home Timeline*.

[10] gRPC: https://grpc.io
[11] RabbitMQ: https://www.rabbitmq.com

3.7 Funcionalidades e requisitos do novo sistema

Nesta secção são referidas funcionalidades e requisitos das características do sistema, nomeadamente, aplicações, serviços e contentores na subsecção 3.7.1, nós e *hosts* na subsecção 3.7.2, regras e métricas na subsecção 3.7.3, monitorização, análise, planeio e execução na subsecção 3.7.4 e a sincronização de dados na subsecção 3.7.5.

3.7.1 Aplicações, serviços e contentores

Estão definidos três tipos de serviços aplicacionais: *frontend*, *backend*, *database*, e dois tipos de contentores: *singleton* e *by_request*.

Os serviços *frontend* dizem respeito aos micro-serviços que implementam um serviço *frontend*, aos quais um cliente pode aceder, através de um balanceador de carga. Os serviços *backend* são micro-serviços que são usados por serviços *frontend*, que implementam diversas funcionalidades necessárias à aplicação. Os serviços *database* são bases de dados que aos quais alguns serviços *backend* dependem para guardar dados. Foi considerado ainda outro tipo de serviço especial: *memcached*, que é uma dependência de alguns micro-serviços, e deve ser lançado sempre na mesma máquina dos serviços, para que seja eficaz.

Os contentores *singleton* dizem respeito aos contentores que contém os componentes de apoio (proxy com autenticação básica, monitor de pedidos, e o prometheus) lançados em todos os nós do sistema. Os gestores garantem que apenas é executado uma réplica de cada componente em cada nó, ao verificar primeiro quais contentores já se encontram a executar no nó, antes de tentar lançar o contentor. Os contentores *by_request* são contentores contendo todos os outros tipos de serviços, incluindo os outros componentes de apoio às operações (gestores locais, agentes kafka, balanceadores de carga, e servidores de registo).

3.7.2 Hosts e nós

Os *hosts* são todas as máquinas controladas pelo sistema, quer façam parte de *clusters* de gestores, ou não. Entre os *hosts* estão as instâncias virtuais iniciadas na *cloud*, e as máquinas *edge* a que o sistema tem acesso.

Um *host* passa a ser um nó do sistema quando este é adicionado a um *docker swarm* de um gestor. Ao ser adicionado a um *swarm*, é configurado com o lançamento de 3 contentores de apoio às operações, que executam em todos os nós: proxy com autenticação básica, monitor de pedidos, e o prometheus.

Para suportar os componentes de apoio iniciados um por região (gestores locais, agentes kafka, balanceadores de carga, servidores de registo), foi preciso definir dois tipos de instâncias virtuais na *cloud*:

- t2.micro. É uma instância com 1 vCPU e 0.5GB de RAM, iniciada para alojar contentores de serviços ou componentes balanceadores de carga.

- t2.medium. É uma instância com 2 vCPU e 4GB de RAM, iniciada para alojar gestores locais, agentes kafka ou servidores de registo, que precisam de mais recursos para funcionarem corretamente.

A instância t2.micro é baseada numa imagem Amazon Machine Images (AMI) pré-configurada com as imagens *docker* do componente de autenticação básica, monitor de pedidos, e o prometheus. Desta forma, não é preciso transferir as imagens *docker* na altura da configuração do nó.

Já a instância t2.medium é baseada numa imagem AMI pré-configurada com as imagens *docker* do componente de autenticação básica, monitor de pedidos, gestores locais, agentes kafka, balanceadores de carga, e servidores de registo. Assim, quaisquer que sejam os componente de apoio iniciados na instância, não é preciso que sejam transferidas as imagens *docker* para se iniciarem os contentores.

3.7.3 Regras e métricas

Foram adicionadas regras e métricas simuladas ao nível de uma aplicação. São aplicadas a todos os contentores que pertençam a essa aplicação. Também foi implementado a possibilidade de associar uma regra a um contentor especifico. Portanto, agora existem 6 tipos de regras: *host* genéricas (aplicadas a todos os nós do sistema), *host* individual (aplicado a um só nó), aplicações individual (aplicadas aos contentores pertencentes à aplicação), serviço genéricas (aplicadas a todos os contentores), serviço individual (aplicada a contentores de um só tipo de serviço), contentores individual (aplicadas a um só contentor). A ordem de aplicação das regras/métricas é a seguinte: aplicação -> serviço -> contentor. Ou seja, regras/métricas ao nível do contentor sobrepõem as definidas ao nível do serviço, e as de serviço sobrepõem as definidas ao nível da aplicação.

3.7.4 Monitorização, análise, planeio e execução

Em relação à monitorização, foi adicionada de uma nova *query* ao prometheus: node_filesystem_avail_bytes. A métrica é usada para ver se o nó tem espaço suficiente no disco para extrair a imagem *docker* que poderá receber, caso tenha que executar um contentor desse serviço.

Os algoritmos de seleção da melhor localização foram alterados para deixarem de usar comparações entre palavras de continentes/regiões/cidades e passarem a usar coordenadas (latência/longitude). Com as coordenadas é possível calcular a distância e obter uma escolha mais genérica e acertada. Para a escolha do melhor *host* é usado um algoritmo (de geometria esférica) que calcula a distância entre as duas coordenadas com até cerca de dez quilómetros de erro. Para a escolha do local de migração de contentores, é usado o algoritmo 1, que calcula a coordenada média com peso, ou seja, tem em conta o número de acessos ao serviço e a sua proveniência. A comparação entre continentes/regiões/cidades era limitada porque havia um número fixo de regiões/continentes, e bastava os nomes

estarem em línguas diferentes (e.g. lisboa/lisbon) para que o algoritmo deixasse de ser eficaz.

3.7.5 Sincronização de dados

Com o uso de vários gestores, tem que existir uma solução de sincronização de dados entre os gestores locais e o gestor principal. Foi escolhida a plataforma distribuída de *streaming* de eventos Kafka, que permite produzir e consumir mensagens em tópicos. Os gestores locais consomem todos os tópicos que o gestor principal produz, e vice-versa. Através do uso das chaves e dos valores dos registos, é possível sincronizar os dados entre os gestores, com o envio para o tópico correspondente, após uma adição, alteração ou remoção numa tabela da base de dados.

3.7.6 Operações assíncronas e paralelismo

Com a introdução do suporte a várias regiões, existe a necessidade de paralelizar certas operações, de modo a diminuir o tempo das chamadas aos vários APIs externos.

Para melhorar o desempenho do sistema, certas operações (descritas nas secções 4.2.8.2 e 4.2.8.3) do gestor principal e dos gestores locais foram implementadas a pensar no assincronismo e paralelismo.

Algumas ações são executadas paralelamente, mas não são assíncronas, ou seja, por exemplo, podem ser enviados vários pedidos a uma API externa, mas a execução da operação pára à espera que todos os pedidos sejam recebidos. Outras ações são assíncronas, no sentido em que a sua execução não bloqueia a operação e a execução de eventuais ações seguintes pode continuar imediatamente depois. A execução pode e deve ser paralela quando existem operações sequenciais que demoram muito tempo a executar e que podem ser executadas em paralelo. Um exemplo disso acontece na configuração de uma instância *cloud* nova. Após ser iniciada a instância, é preciso iniciar dois contentores, presentes em todos os *hosts*: prometheus e o monitor de pedidos, que podem ser lançados em paralelo, porque não se relacionam entre si. Mas é preciso esperar que ambos iniciem para se poder afirmar que a instância está totalmente configurada. Ou seja, é uma operação paralela, mas não assíncrona.

Outras operações são paralelas e assíncronas, como é o caso do término das instâncias *cloud* quando o sistema é terminado. A operação pode ser paralela porque os pedidos para terminar as instâncias podem ser enviados em simultâneo, e pode ser assíncrona porque não é preciso esperar que o pedido seja executado pela Amazon.

3.7.7 Ambiente de execução do sistema

Como o sistema é para ser usado em máquinas não proprietárias, deve ser possível distinguir os contentores e nós geridos pelo sistema de gestão, de outros contentores e nós

exterior ao sistema. Uma solução possível, é apenas considerar contentores e nós que estejam marcados, por exemplo, através de *tags* ou *labels*, definidos pelo sistema.

Capítulo 4

Implementação

O capitulo sobre a implementação descreve, na secção 4.1, as tecnologia utilizadas, e na secção 4.2 são detalhados todos os aspetos da implementação de todos os componentes do sistema.

4.1 Tecnologias utilizadas

Durante o desenvolvimento do sistema foram utilizadas várias linguagens 4.1.1 para a implementação dos componentes e para a adaptação de micro-serviços, vários tipos de gestão de dados 4.1.2, foram incluídas certas bibliotecas e *frameworks* 4.1.3 para ajudar o processo de desenvolvimento, usadas ferramentas 4.1.4, quer para implementação, quer para *debugging*, e foram usados certos produtos 4.1.5 como parte fundamental do sistema.

4.1.1 Linguagens

Como linguagens foi usado o Java 11 na implementação do gestor principal e gestores locais, incluindo as bibliotecas da base de dados e dos serviços. A linguagem Go, na versão 1.14.6, foi usado no cliente de registo e no monitor de pedidos, por ser uma linguagem leve e com funcionalidades que permitem execução assíncrona, visto que, no caso do monitor de pedidos, são integrados em todos os nós do sistema, e no caso do cliente de registo é incluído juntamente com cada contentor lançado. A *framework* React+Typescript foi usada na implementação do Hub de gestão, o que envolveu o uso de Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) e Javascript. Foi escolhido o React por ser uma *framework* que facilita a implementação de uma aplicação *web* reativa aos dados e estado do sistema.

O sistema inclui aplicações compostas por micro-serviços implementados em Java, Go, Python, C++, C e NodeJS. Embora não tenha sido necessário implementar nenhum

micro-serviço de raiz, vários micro-serviços tiveram que ser alterados para usarem a funcionalidade de descoberta de serviços incluída no sistema, o que implicou mudanças no seu código.

4.1.2 Base de dados

O gestor principal usa o motor de base de dados H2 [1] para guardar os seus dados, sendo estes guardados permanentemente num ficheiro no disco. Já um gestor local usa também H2, mas os dados são guardados em memória, visto que não precisam de ser persistentes porque são obtidos através do Kafka.

Certos micro-serviços usam Redis [2] ou MongoDB [3] para a persistência de dados. As bases de dados de cada micro-serviço são lançadas num contentor juntamente com o contentor do micro-serviço. Alguns micro-serviços também usam o Memcached [4], para aumentar a velocidade de acesso aos dados, através da funcionalidade de *caching* em memória.

4.1.3 Bibliotecas e frameworks

As principais bibliotecas e *frameworks* usadas no desenvolvimento dos componentes foram:


Spring Boot - O Spring Boot permite criar aplicações Spring [5] pré-configuradas, facilitando o desenvolvimento de certas partes da aplicação. Spring é uma *framework* usada no desenvolvimento de aplicações Java, e foi incluída na implementação dos gestores.


Project Lombok - Lombok é uma biblioteca que, através do processamento de anotações, gera automaticamente código que muitas vezes é repetitivo na linguagem Java. Por exemplo, é capaz de gerar automaticamente *setters* e *getters* para as propriedades, os construtores, ou métodos *equals*, *toString* e *hashCode*, de um objeto.


MapStruct - MapStruct é outra biblioteca que gera código automaticamente através do processamento de anotações, e deteção automática das propriedades das classes. Permite mapear classes Java em outras classes Java, e foi usado para fazer o mapeamento entre as entidades JPA da base de dados e os DTOs, transferidos entre os gestores.

[1] Motor de base de dados H2: https://www.h2database.com
[2] Redis: https://redis.io
[3] MongoDB: https://www.mongodb.com
[4] Memcached: https://memcached.org
[5] Spring: https://spring.io

 Docker Spotify Java Client - Foi usado o cliente Java para o Docker criado pela Spotify, tanto no gestor principal, como no gestor local. O que permite usar as funcionalidades do Docker numa aplicação Java.

 ReactJS - React foi a *framework* de Javascript usada para a implementação do Hub de gestão. React promove a definição de componentes, que depois podem ser usados em várias partes da aplicação, o que ajuda a evitar repetição, que muitas vezes é um problema no desenho de interfaces gráficas usando apenas HTML. React recarrega apenas as partes da página que sofrem alterações devido à mudança de estado da aplicação *web*, o que é mais eficiente do que a tradicional aplicação implementada em HTML e Javascript puro, que precisa de recarregar toda a página para se fazerem notar as alterações.

 Typescript - O TypeScript permite tipificar código Javascript, o que permitiu tornar o Hub de gestão menos susceptível a erros de execução, ao detetar os eventuais erros na altura da compilação do código através da analise de tipos.

 React-redux - O React-redux foi usado para controlar os dados do Hub de gestão que são obtidos dos gestores. Usando a biblioteca, é possível ter acesso fácil aos dados em qualquer componente, através do uso de funções que ligam as propriedades do componente aos dados obtidos nos gestores.

 Materialize CSS - O Materialize CSS é uma *framework* baseada no Material Design [6] que inclui componentes Javascript e elementos CSS para serem usados numa aplicação React. O Hub de gestão inclui certos elementos Javascript da *framework* Materialize CSS: *buttons*, *breadcrumbs*, *cards*, *collections*, *footer*, *icons*, *navbar*, *pagination*, *preloader*. Bem como algumas definições CSS, embora bastante customizadas para dar um estilo único ao Hub de gestão.

 Sass - Sass é uma extensão do CSS que introduz variáveis, *nesting*, *mixins*, herança, entre outras funcionalidades, para ajudar na escrita de folhas de estilo. Foi usada no Hub de gestão para definir o estilo visual da aplicação, e permitir definir facilmente dois modos de apresentação, escuro e claro.

 React Simple Maps - O RSM usa d3-geo [7] e TopoJSON [8] para, através de uma API declarativa, criar mapas Scalable Vector Graphics (SVG). Foi usado no Hub de

[6] Material Design: https://material.io/design
[7] d3-geo: https://github.com/d3/d3-geo
[8] TopoJSON: https://github.com/topojson/topojson

gestão para criar os mapas interativos presentes na página principal, bem como em outras páginas para ver e/ou selecionar coordenadas.

 Gorilla Mux - Gorilla Mux é um pacote escrito em Go que implementa um *router* de pedidos, associando *endpoints* ao método que processa o respetivo pedido. Foi usado nos componentes cliente de registo e monitor de pedidos, para expor APIs REST.

4.1.4 Ferramentas

O conjunto de ferramentas usadas no desenvolvimento do sistema foram:

 IntelliJ IDEA - IntelliJ foi o Integrated development environment (IDE) usado para a escrita do código de todos os componentes do sistema.

 kafkacat - O kafkacat é uma ferramenta capaz de produzir e consumir tópicos Kafka. Foi usado maioritariamente para *debugging* da troca de dados entre os gestores.

 Postman - Postman é uma ferramenta usada no desenvolvimento de APIs. Foi usada para testar os APIs que os vários componentes do sistema expõem.

 Json Formatter - Json Formatter é uma extensão do Chrome [9] que formata mensagens JSON visualizadas diretamente no *browser*, para facilitar a sua leitura.

 Checkstyle - Foi definido um ficheiro Checkstyle para formatar automaticamente o código escrito na linguagem Java, para ficar uniforme em todos os componentes Java do projeto.

 React Developer Tools - React Developer Tools é uma extensão do Chrome que adiciona ferramentas de *debugging* de uma aplicação React ao *browser*.

 Golang playground - Go Playground é um serviço *web* que permite executar código Go através de um *browser*. Foi usado principalmente para experimentar código Go.

 npm - npm foi usado para instalar e gerir pacotes de bibliotecas no Hub de gestão.

[9] Chrome browser: http://google.com/chrome

 Maven - Maven foi usado para instalar e gerir dependências dos componentes escritos em Java, bem como construir o respetivo ficheiro Java ARchive (JAR).

4.1.5 Produtos

Os principais produtos incluídos no sistema foram:

 Docker - Docker é uma plataforma para criar e executar código dentro de contentores. Foram usados Dockerfiles [10] para definir a construção de imagens docker, guardadas em repositórios no Docker Hub, que depois são usadas para iniciar os contentores. Cada gestor começa e gere um *swarm*, através do *docker swarm* [11], para fazer a gestão dos nós que controla.

 Kafka - Kafka foi a plataforma de *streaming* de eventos usada para propagar os dados entre os vários gestores.

 Drools - Foram usados *templates* Drools para implementar as regras aos *hosts*, aplicações, serviços, e contentores, definidas no sistema.

4.2 Sistema de gestão dinâmico 

O sistema de gestão dinâmico implementa a arquitetura descrita na secção 3.2, e está dividido em:

- Duas bibliotecas 4.2.1: uma para definir a base de dados usada pelos gestores, outra com código que implementa os serviços comuns ao gestor principal e ao gestor local;
- Três componentes principais: Gestor principal 4.2.2.1, Gestor local 4.2.2.2 e Hub de gestão 4.2.2.3;
- Sete componentes de apoio às operações: Proxy com autenticação básica 4.2.4.1, Servidor de registo 4.2.4.2, Cliente de registo 4.2.4.3, Monitor de pedidos de localização 4.2.4.4, Balanceador de carga 4.2.4.5, Gestor do balanceador de carga 4.2.4.6, Prometheus 4.2.4.7 e agentes Kafka 4.2.4.8.

4.2.1 Bibliotecas dos gestores

Foram desenvolvidas duas bibliotecas Java com código contendo as definições das entidades e repositórios da base de dados, bem como os serviços utilizados, tanto pelo gestor principal, como pelos gestores locais. As bibliotecas permitem partilhar código entre

[10] Dockerfile: https://docs.docker.com/engine/reference/builder
[11] Docker swarm: https://docs.docker.com/engine/swarm

o gestor principal e o gestor local, visto que existem funcionalidades que estão presentes em ambos. Antes da compilação dos gestores, as bibliotecas são compactadas em ficheiros JAR, ficando disponíveis para serem importados como uma dependência, por exemplo, num ficheiro *pom.xml* usado pela ferramenta de compilação *maven*[12], como na listagem 4.1.

Listagem 4.1: XML para importar o código da definição da base de dados e dos serviços usado nos gestores.

```
<dependency>
    <groupId>pt.unl.fct.miei.usmanagement.manager</groupId>
    <artifactId>manager-database</artifactId>
    <version>0.0.1</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>pt.unl.fct.miei.usmanagement.manager</groupId>
    <artifactId>manager-services</artifactId>
    <version>0.0.1</version>
    <scope>compile</scope>
</dependency>
```

A biblioteca da base de dados define um total de 49 entidades JPA que são traduzidas em 62 tabelas na base de dados, após serem processadas pela *framework* Spring [13]. As tabelas são usadas para guardar os dados necessários à execução dos gestores, como: aplicações, serviços, contentores, instâncias virtuais na *cloud*, máquinas *edge*, nós, condições, regras, métricas simuladas, componentes de apoio lançados, *heartbeats*, e todos os dados de relações entre entidades.

Já a biblioteca com o código dos serviços, inclui 41 serviços que implementam as funcionalidades comuns aos gestores, desde código para gerir o *docker swarm*, gerir e configurar *hosts* e nós, iniciar contentores, executar comandos SSH ou *bash*, ou listar/adicionar/alterar/remover entidades da base de dados.

4.2.2 Componentes principais

Como componentes principais, o sistema inclui um gestor principal, que tem a visão completa do sistema, gere os gestores locais e todos os componentes de apoio. Inclui um gestor local, para ser lançado mais perto dos clientes e permitir a separação de responsabilidades entre as várias regiões suportadas. E uma aplicação *web*, o Hub de gestão, para a interação manual e visualização do sistema através de uma interface gráfica.

[12] Maven: https://maven.apache.org
[13] Spring: https://spring.io

4.2.2.1 Gestor principal

O gestor principal é implementado em Java, com uso do *framework* Spring Boot [14], baseado no gestor feito no trabalho anterior. É o componente que faz a gestão principal do sistema, sendo o ponto central que guarda a definição de aplicações, serviços, regras e métricas simuladas, e gere os componentes (gestores locais, servidores de registo, balanceadores de carga e agentes kafka) lançados em cada região.

No inicio da sua execução, configura o sistema inicial:

1. Executa um *script* para introduzir certos valores por defeito na base de dados, e limpa os dados relativos a uma eventual execução anterior:
   - Adiciona um utilizador com permissões de administrador;
   - Adiciona a informação sobre os componentes de apoio;
   - Define todas as aplicações, serviços e respetivas dependências, enunciadas nos casos de estudo, na secção 5.3;
   - Adiciona os tipos de componentes (*host*, *app*, *service*, *container*) geridos pelo sistema;
   - Adiciona os operadores suportados: diferente (!=), igual (==), maior (>), menor (<), maior ou igual (>=), menor ou igual (<=);
   - Adiciona as decisões possíveis dos hosts: *none*, *overwork*, *underwork*; e dos serviços: *none*, *stop*, *replicate*, *migrate*;
   - Adiciona os campos suportados para formar condições: *cpu*, *cpu-%*, *ram*, *ram-%*, *rx-bytes*, *tx-bytes*, *rx-bytes-per-second*, *tx-bytes-per-second*, *latency*, *bandwidth*, *bandwidth-%*, *filesystem-available-space*;
   - Adiciona os tipos de valores suportados para formar condições: *effective-value*, *average-value*, *deviation-%-on-average-value*, *deviation-%-on-last-value*;
   - Define três condições: *cpu-%* > 90%, *ram-%* > 90% e *rx-bytes* > 500000;
   - Apaga certos dados sobre a execução anterior: eventos de *hosts* e de serviços, monitorização sobre *hosts* e sobre serviços, dados sobre balanceadores de carga, servidores de registo, gestores locais e agentes kafka que estiveram a executar, dados de contentores e nós, endereços IP elásticos, configurações e *heartbeats*.
2. Configura o *docker swarm* inicial, sendo o nó principal o *host* onde é lançado.
3. Aloca endereços IP elástico na AWS, um por cada região suportada, para, durante a execução, serem associados cada um a uma instância *cloud* onde é lançado um servidor de registo, como explicado na secção 4.2.4.2.

[14] Spring Boot: https://spring.io/projects/spring-boot

4. Inicia três ciclos periódicos para a sincronização das instâncias na *cloud*, contentores e nós guardados na base de dados, comparativamente com o estado do seu *docker swarm* e dos *heartbeats* enviados pelos gestores locais. Mais informação disponível na secção 4.2.3.

Também inicia um servidor HTTP disponibilizando um API, visível na tabela 4.1, para o Hub de gestão interagir com o gestor principal. Através do API, o Hub de gestão pode comunicar com o gestor principal para obter toda a informação necessária para mostrar o estado do sistema ao utilizador. Bem como adicionar, alterar e apagar aplicações, serviços, regras e métricas simuladas. E executar ações para parar contentores, modificar o estado do *swarm* de nós, lançar e parar os componentes do sistema, ver os registos, e executar comandos SSH.

Tabela 4.1: API do gestor principal. *Endpoints* relativos a /api.

Pedido	**Endpoint**	**Descrição**
GET	/basicauth	Usado para autenticação.
GET	/local-managers	Obtém a lista de gestores locais a executarem.
GET	/local-managers/{id}	Obtém o gestor local com id {id}.
POST	/local-managers	Inicia um gestor local numa região, ou num endereço especifico, passado no corpo do pedido.
DELETE	/local-managers/{id}	Pára o gestor local com id {id}.
GET	/registration-servers	Obtém a lista de servidores de registo a executarem.
GET	/registration-servers/{id}	Obtém o servidor de registo com id {id}.
POST	/registration-servers	Inicia um servidor de registo numa região, ou num endereço especifico, passado no corpo do pedido.
DELETE	/registration-servers/{id}	Pára o servidor de registo com id {id}.
GET	/load-balancers	Obtém a lista de balanceadores de carga a executarem.
GET	/load-balancers/{id}	Obtém o balanceador de carga com id {id}.
POST	/load-balancers	Inicia um balanceador de carga numa região, ou num endereço especifico, passado no corpo do pedido.
DELETE	/load-balancers/{id}	Pára o balanceador de carga com id {id}.
GET	/kafka	Obtém a lista de agentes kafka a executarem.
GET	/kafka/{id}	Obtém o agente kafka com id {id}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/kafka	Inicia um agente kafka numa região, ou num endereço especifico, passado no corpo do pedido.
DELETE	/kafka/{id}	Pára o agente kafka com id {id}.
GET	/apps	Obtém a lista de aplicações.
GET	/apps/{name}	Obtém a aplicação com nome {name}.
POST	/apps	Adiciona a aplicação passada no corpo do pedido.
PUT	/apps/{name}	Altera a aplicação com nome {name}, com os valores passados no corpo do pedido.
DELETE	/apps/{name}	Apaga a aplicação com nome {name}.
GET	/apps/{name}/services	Obtém a lista de serviços da aplicação com nome {name}.
POST	/apps/{name}/services	Associa os serviços passados no corpo do pedido à aplicação com nome {name}.
DELETE	/apps/{name}/services	Desassocia os serviços passados no corpo do pedido da aplicação com nome {name}.
POST	/apps{name}/launch	Lança todos os serviços da aplicação com nome {name}, nas coordenadas passada no corpo do pedido.
GET	/services	Obtém a lista serviços.
GET	/services/{name}	Obtém o serviço com nome {name}.
POST	/services	Adiciona o serviço contido no corpo do pedido.
PUT	/services/{name}	Atualiza o serviço com nome {name}, com os valores no corpo do pedido.
DELETE	/services/{name}	Remove o serviço com nome {name}.
GET	/services/{name}/apps	Obtém a lista de aplicações associadas ao serviço com nome {name}.
POST	/services/{name}/apps	Associa as aplicações passadas no corpo do pedido, ao serviço com nome {name}.
DELETE	/services/{serviceName}/apps/{appName}	Desassocia a aplicação com nome {appName} do serviço com nome {serviceName}.
GET	/services/{name}/dependencies	Obtém a lista das dependências do serviço com nome {name}.
POST	/services/{name}/dependencies	Adiciona as dependências passadas no corpo do pedido, ao serviço com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
DELETE	/services/{serviceName}/ dependencies	Remove as dependências passadas no corpo do pedido, ao serviço com nome {name} do serviço com nome {name}.
GET	/services/{name}/ dependents	Obtém a lista dos dependentes ao serviço com nome {name}.
GET	/services/{name}/ predictions	Obtém as previsões feitas ao serviço com nome {name}.
POST	/services/{name}/ predictions	Adiciona uma previsão, passada no corpo do pedido, ao serviço com nome {name}.
DELETE	/services/{name}/ predictions	Remove todas as previsões contidas no corpo do pedido, feitas ao serviço com nome {name}.
DELETE	/services/{serviceName}/ predictions/ {predictionName}	Remove a previsão com nome {predictionName} ao serviço com nome {serviceName}.
GET	/containers	Obtém a lista de todos os contentores do sistema.
GET	/containers/{id}	Obtém o contentor com id {id}.
POST	/containers/sync	Sincroniza os dados sobre contentores guardados na base de dados, com a informação dos *docker swarms* de todos os gestores.
GET	/containers/{id}/logs	Obtém os registos do contentor com id {id}.
GET	/hosts/edge	Obtém a lista de hosts na periferia da rede.
GET	/hosts/edge/{publicIp}/ {privateIp}	Obtém o host na periferia da rede com o ip público {publicIp} e ip privado {privateId}.
POST	/hosts/edge	Adiciona o host contido no corpo do pedido.
PUT	/hosts/edge/{publicIp}/ {privateIp}	Altera o host com ip público {publicIp} e ip privado {privateId}, com os valores passados no corpo do pedido.
DELETE	/hosts/edge/{publicIp}/ {privateIp}	Apaga o host com ip público {publicIp} e ip privado {privateId}.
GET	/hosts/cloud	Obtém a lista de instâncias a executar na cloud.
GET	/hosts/cloud/{instanceId}	Obtém a instância na cloud com id {instanceId}.
POST	/hosts/cloud	Inicia e configura uma instância perto das coordenadas passadas no corpo do pedido.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
PUT	/hosts/cloud/ {instanceId}/start	Inicia a instância com id {instanceId}.
PUT	/hosts/cloud/ {instanceId}/stop	Pára a instância com id {instanceId}.
DELETE	/hosts/cloud/{instanceId}/ terminate	Termina a instância com id {instanceId}.
POST	/hosts/cloud/sync	Executa uma sincronização das instâncias cloud conhecidas com a informação na AWS.
GET	/hosts/cloud/regions	Obtém a lista de regiões AWS suportadas.
GET	/nodes	Obtém a lista de nós de todo o sistema.
GET	/nodes/{id}	Obtém o nó com id {id}.
POST	/nodes	Inicia um nó no gestor principal, ou local, perto da coordenada, ou num nó especifico, passado no corpo do pedido.
PUT	/nodes/{id}	Atualiza o estado do nó com id {id} com os valores passados no corpo do pedido.
DELETE	/nodes/{id}	Remove o nó com id {id}.
PUT	/nodes/{publicIp}/ {privateIp}	Remove o nó presente no host com ip público {publicIp} e ip privado {privateIp}.
POST	/nodes/{id}/join	Faz com que o nó com {id} volte a entrar no *swarm*.
POST	/nodes/sync	Sincroniza os dados sobre nós guardados na base de dados, com a informação dos *docker swarms* de todos os gestores.
GET	/fields	Obtém a lista de campos, usados para construir condições.
GET	/fields/{name}	Obtém o campo com nome {name}.
GET	/component-types	Obtém a lista dos tipos de componentes, usada nas regras e métricas simuladas.
GET	/component-types/{name}	Obtém o tipo de componente com nome {name}.
GET	/rules/conditions	Obtém a lista das condições definidas.
GET	/rules/conditions/{name}	Obtém a condição com nome {name}.
POST	/rules/conditions	Adiciona a condição passada no corpo do pedido.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
PUT	/rules/conditions/{name}	Altera a condição com nome {name}, com os valores passados no corpo do pedido.
DELETE	/rules/conditions/{name}	Apaga a condição com nome {name}.
GET	/rules/conditions/hosts	Obtém a lista de todas as associações entre condições e regras de hosts.
GET	/rules/conditions/{name}/hosts	Obtém a lista de regras de hosts associadas à condição com nome {name}.
GET	/rules/conditions/apps	Obtém a lista de todas as associações entre condições e regras de aplicações.
GET	/rules/conditions/{name}/apps	Obtém a lista de regras de aplicações associadas à condição com nome {name}.
GET	/rules/conditions/services	Obtém a lista de todas as associações entre condições e regras de serviços.
GET	/rules/conditions/{name}/services	Obtém a lista de regras de serviços associadas à condição com nome {name}.
GET	/rules/conditions/containers	Obtém a lista de todas as associações entre condições e regras de contentores.
GET	/rules/conditions/{name}/containers	Obtém a lista de regras de contentores associadas à condição com nome {name}.
GET	/rules/hosts	Obtém a lista de regras de hosts.
GET	/rules/hosts/{name}	Obtém a regra de host com nome {name}.
POST	/rules/hosts	Adiciona a regra de host contida no corpo do pedido.
PUT	/rules/hosts/{name}	Atualiza a regra de host com nome {name} com os valores passados no corpo do pedido.
DELETE	/rules/hosts/{name}	Apaga a regra de host com nome {name}.
GET	/rules/hosts/{name}/conditions	Obtém a lista de condições da regra de host com nome {name}.
POST	/rules/hosts/{name}/conditions	Associa as condições passadas no corpo do pedido, à regra de host com nome {name}.
DELETE	/rules/hosts/{name}/conditions	Desassocia as condições passadas no corpo do pedido, da regra de host com nome {name}.
GET	/rules/hosts/{name}/cloud-hosts	Obtém a lista de hosts na cloud associados à regra de host com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/rules/hosts/ {name}/cloud-hosts	Associa os hosts na cloud passados no corpo do pedido, à regra de host com nome {name}.
DELETE	/rules/hosts/ {name}/cloud-hosts	Desassocia os hosts na cloud passados no corpo do pedido, da regra de host com nome {name}.
GET	/rules/hosts/ {name}/edge-hosts	Obtém a lista de hosts na edge associados à regra de host com nome {name}.
POST	/rules/hosts/ {name}/edge-hosts	Associa os hosts na edge passados no corpo do pedido, à regra de host com nome {name}.
DELETE	/rules/hosts/ {name}/edge-hosts	Desassocia os hosts na edge passados no corpo do pedido, da regra de host com nome {name}.
GET	/rules/apps	Obtém a lista de regras de aplicações.
GET	/rules/apps/{name}	Obtém a regra de aplicação com nome {name}.
POST	/rules/apps	Adiciona a regra de aplicação contida no corpo do pedido.
PUT	/rules/apps/{name}	Atualiza a regra de aplicação com nome {name} com os valores passados no corpo do pedido.
DELETE	/rules/apps/{name}	Apaga a regra de aplicação com nome {name}.
GET	/rules/apps/{name}/ conditions	Obtém a lista de condições da regra de aplicação com nome {name}.
POST	/rules/apps/{name}/ conditions	Associa as condições passadas no corpo do pedido, à regra de aplicação com nome {name}.
DELETE	/rules/apps/{name}/ conditions	Desassocia as condições passadas no corpo do pedido, da regra de aplicação com nome {name}.
GET	/rules/apps/{name}/ apps	Obtém a lista de aplicações associados à regra de aplicação com nome {name}.
POST	/rules/apps/{name}/ apps	Associa as aplicações passados no corpo do pedido, à regra de aplicação com nome {name}.
DELETE	/rules/apps/{name}/ apps	Desassocia as aplicações passados no corpo do pedido, da regra de aplicação com nome {name}.
GET	/rules/services	Obtém a lista de regras de serviços.
GET	/rules/services/{name}	Obtém a regra de serviço com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/rules/services	Adiciona a regra de serviço contida no corpo do pedido.
PUT	/rules/services/{name}	Atualiza a regra de serviço com nome {name} com os valores passados no corpo do pedido.
DELETE	/rules/services/{name}	Apaga a regra de serviço com nome {name}.
GET	/rules/services/{name}/conditions	Obtém a lista de condições da regra de serviço com nome {name}.
POST	/rules/services/{name}/conditions	Associa as condições passadas no corpo do pedido, à regra de serviço com nome {name}.
DELETE	/rules/services/{name}/conditions	Desassocia as condições passadas no corpo do pedido, da regra de serviço com nome {name}.
GET	/rules/services/{name}/services	Obtém a lista de serviços associados à regra de serviço com nome {name}.
POST	/rules/services/{name}/services	Associa os serviços passados no corpo do pedido, à regra de serviço com nome {name}.
DELETE	/rules/services/{name}/services	Desassocia os serviços passados no corpo do pedido, da regra de serviço com nome {name}.
GET	/rules/containers	Obtém a lista de regras de contentores.
GET	/rules/containers/{name}	Obtém a regra de contentor com nome {name}.
POST	/rules/containers	Adiciona a regra de contentor contida no corpo do pedido.
PUT	/rules/containers/{name}	Atualiza a regra de contentor com nome {name} com os valores passados no corpo do pedido.
DELETE	/rules/containers/{name}	Apaga a regra de contentor com nome {name}.
GET	/rules/containers/{name}/conditions	Obtém a lista de condições da regra de contentor com nome {name}.
POST	/rules/containers/{name}/conditions	Associa as condições passadas no corpo do pedido, à regra de contentor com nome {name}.
DELETE	/rules/containers/{name}/conditions	Desassocia as condições passadas no corpo do pedido, da regra de contentor com nome {name}.
GET	/rules/containers/{name}/containers	Obtém a lista de contentores associados à regra de contentor com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/rules/containers/ {name}/containers	Associa os contentores passados no corpo do pedido, à regra de contentor com nome {name}.
DELETE	/rules/containers/ {name}/containers	Desassocia os contentores passados no corpo do pedido, da regra de contentor com nome {name}.
GET	/decisions	Obtém a lista de decisões possíveis.
GET	/decisions/services	Obtém a lista das decisões possíveis para serviços.
GET	/decisions/hosts	Obtém a lista das decisões possíveis para hosts.
GET	/operators	Obtém a lista de operadores que podem ser usados para formar condições.
GET	/operators/{name}	Obtém o operador com nome {name}.
GET	/operators/{name}/ conditions	Obtém a lista de condições que usam o operador com nome {name}.
GET	/simulated-metrics/hosts	Obtém a lista de métricas simuladas definidas para hosts.
GET	/simulated-metrics/hosts/ {name}	Obtém a métrica simulada para hosts com nome {name}.
POST	/simulated-metrics/hosts	Adiciona a métrica simulada para hosts definida no corpo do pedido.
PUT	/simulated-metrics/hosts/ {name}	Altera a métrica simulada para hosts com nome {name}, com os valores passados no corpo do pedido.
DELETE	/simulated-metrics/hosts/ {name}	Apaga a métrica simulada para hosts com nome {name}.
GET	/simulated-metrics/hosts/ {name}/cloud-hosts	Obtém a lista de hosts na cloud que usam a métrica simulada com nome {name}.
POST	/simulated-metrics/hosts/ {name}/cloud-hosts	Associa a métrica simulada com nome {name} aos hosts na cloud passados no corpo do pedido.
DELETE	/simulated-metrics/hosts/ {name}/cloud-hosts	Desassocia a métrica simulada com nome {name} aos hosts na cloud passados no corpo do pedido.
GET	/simulated-metrics/hosts/ {name}/edge-hosts	Obtém a lista de hosts na edge que usam a métrica simulada com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/simulated-metrics/hosts/{name}/edge-hosts	Associa a métrica simulada com nome {name} aos hosts na edge passados no corpo do pedido.
DELETE	/simulated-metrics/hosts/{name}/edge-hosts	Desassocia a métrica simulada com nome {name} aos hosts na edge passados no corpo do pedido.
GET	/simulated-metrics/apps	Obtém a lista de métricas simuladas definidas para aplicações.
GET	/simulated-metrics/apps/{name}	Obtém a métrica simulada para aplicações com nome {name}.
POST	/simulated-metrics/apps	Adiciona a métrica simulada para aplicações definida no corpo do pedido.
PUT	/simulated-metrics/apps/{name}	Altera a métrica simulada para aplicações com nome {name}, com os valores passados no corpo do pedido.
DELETE	/simulated-metrics/apps/{name}	Apaga a métrica simulada para aplicações com nome {name}.
GET	/simulated-metrics/apps/{name}/apps	Obtém a lista de aplicações que usam a métrica simulada com nome {name}.
POST	/simulated-metrics/apps/{name}/apps	Associa a métrica simulada com nome {name} às aplicações passados no corpo do pedido.
DELETE	/simulated-metrics/apps/{name}/apps	Desassocia a métrica simulada com nome {name} às aplicações passados no corpo do pedido.
GET	/simulated-metrics/services	Obtém a lista de métricas simuladas definidas para serviços.
GET	/simulated-metrics/services/{name}	Obtém a métrica simulada para serviços com nome {name}.
POST	/simulated-metrics/services	Adiciona a métrica simulada para serviços definida no corpo do pedido.
PUT	/simulated-metrics/services/{name}	Altera a métrica simulada para serviços com nome {name}, com os valores passados no corpo do pedido.
DELETE	/simulated-metrics/services/{name}	Apaga a métrica simulada para serviços com nome {name}.
GET	/simulated-metrics/services/{name}/services	Obtém a lista de serviços que usam a métrica simulada com nome {name}.

Continua na próxima página

Tabela 4.1 (continuação)

Pedido	Endpoint	Descrição
POST	/simulated-metrics/services/{name}/services	Associa a métrica simulada com nome {name} aos serviços passados no corpo do pedido.
DELETE	/simulated-metrics/services/{name}/services	Desassocia a métrica simulada com nome {name} aos serviços passados no corpo do pedido.
GET	/simulated-metrics/containers	Obtém a lista de métricas simuladas definidas para contentores.
GET	/simulated-metrics/containers/{name}	Obtém a métrica simulada para contentores com nome {name}.
POST	/simulated-metrics/containers	Adiciona a métrica simulada para contentores definida no corpo do pedido.
PUT	/simulated-metrics/containers/{name}	Altera a métrica simulada para contentores com nome {name}, com os valores passados no corpo do pedido.
DELETE	/simulated-metrics/containers/{name}	Apaga a métrica simulada para contentores com nome {name}.
GET	/simulated-metrics/containers/{name}/containers	Obtém a lista de contentores que usam a métrica simulada com nome {name}.
POST	/simulated-metrics/containers/{name}/containers	Associa a métrica simulada com nome {name} aos contentores passados no corpo do pedido.
DELETE	/simulated-metrics/containers/{name}/containers	Desassocia a métrica simulada com nome {name} aos contentores passados no corpo do pedido.
GET	/regions	Obtém a lista de regiões.
GET	/region/{name}	Obtém a região com nome {name}.
POST	/ssh/execute	Executa o comando SSH passado no corpo do pedido.
GET	/ssh/scripts	Obtém a lista de *scripts* suportados, que podem ser carregados.
POST	/ssh/upload	Carrega o ficheiro passado no corpo do pedido.
GET	/logs	Obtém a lista de registos.

Ao ser terminado, pára de consumir e produzir registos *kafka*, pára todos os contentores do sistema, incluindo todos os componentes de apoio, de-aloca todos os endereços IP elásticos na AWS, termina todas as instâncias virtuais iniciadas na AWS e destrói o seu

docker swarm.

Base de dados

Os dados do gestor principal são guardados numa base de dados H2 em disco, com o esquema baseado na biblioteca da base de dados indicado anteriormente na secção 4.2.1. Os valores iniciais são carregados através de um *script* que define vários valores por defeito, introduz as aplicações enunciadas na secção 5.3, incluindo todos os seus serviços e respetivas dependências.

Ciclo de sincronização da base de dados com o *docker swarm*

A sincronização dos nós e contentores da base de dados com o *docker swarm* é feita através de um ciclo periódico que executa de 10 em 10 segundos.

O objetivo do ciclo de sincronização é manter os dados na base de dados em conformidade com a informação do *docker swarm*, em casos, por exemplo, quando um contentor aborta e não fica mais disponível, ou quando um nó deixa de estar disponível por alguma razão. Ao guardar os dados na base de dados, é possível detetar alterações ao estado do *docker swarm*, e comunicá-las aos gestores locais.

Mais detalhes sobre a sincronização de dados estão disponíveis na secção 4.2.3.

Registos

Para que os registos do gestor principal fiquem disponíveis para visualização no Hub de gestão, foi configurado um *appender* (listagem 4.2) ao LOGBack [15], que foi o *framework* usado para registar a atividade dos gestores, para que o registos sejam inseridos em tabelas na base de dados.

Listagem 4.2: Configuração do DB appender do LOGBack.

```
<appender name="DB" class="ch.qos.logback.classic.db.DBAppender">
    <connectionSource class="ch.qos.logback.core.db.DriverManagerConnectionSource">
        <driverClass>${spring.datasource.driverClassName}</driverClass>
        <url>${spring.datasource.url}</url>
        <user>${spring.datasource.username}</user>
        <password>${spring.datasource.password}</password>
    </connectionSource>
</appender>
```

Com os registos na base de dados, estes são acedidos pelo Hub de gestão através do API definido pelo gestor principal.

Segurança

O gestor principal está protegido por autenticação básica, sendo o único utilizador autorizado aquele que foi introduzido no *script* de iniciação da base de dados, anteriormente

[15] LOGBack: http://logback.qos.ch/

explicado. A configuração das definições de segurança foi feita através do Spring security [16], visível na listagem 4.3. Quanto à autorização, todos os pedidos, para além do *login* e *logout*, têm que incluir o *header* Authorization, com a informação sobre o utilizador e a palavra-passe. Em relação ao Cross-origin resource sharing (CORS), como não é conhecido o Uniform Resource Locator (URL) do Hub de gestão porque o seu *deployment* não foi feito, todas as fontes têm que ser aceites, através do *wild card* *.

Listagem 4.3: Configuração de segurança do gestor principal, através do Spring security.

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .headers().frameOptions().sameOrigin()
        .and().authorizeRequests().anyRequest().authenticated()
        .and().formLogin().permitAll()
        .and().logout().permitAll()
        .and().httpBasic()
        .and().cors();
}

public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration corsConfiguration = new CorsConfiguration();
    corsConfiguration.setAllowedOrigins(List.of("*"));
    corsConfiguration.setAllowedMethods(List.of("HEAD", "GET", "POST", "PUT", "DELETE",
        "PATCH"));
    corsConfiguration.setAllowCredentials(true);
    corsConfiguration.setAllowedHeaders(List.of("Authorization", "Cache-Control",
        "Content-Type"));
    UrlBasedCorsConfigurationSource corsConfigurationSource =
        new UrlBasedCorsConfigurationSource();
    corsConfigurationSource.registerCorsConfiguration("/**", corsConfiguration);
    return corsConfigurationSource;
}
```

4.2.2.2 Gestor local

O gestor local é implementado em Java, com uso do *framework* Spring boot, baseado no gestor feito no trabalho anterior. É lançado num contentor pelo gestor principal, até no máximo um por cada região suportada. Como o componente usa o *docker* num próprio contentor *docker*, teve que ser configurado para tal. O ficheiro /var/run/docker.sock do *host* é montado no mesmo ficheiro no contentor. Assim, o contentor e o *host* onde executa partilham o mesmo *socket Unix* do *Docker engine*.

O gestor local precisa de três valores para iniciar a sua execução, que são passados como variáveis de ambiente do contentor, pelo gestor principal:

[16] Spring security: https://spring.io/projects/spring-security

1. **id** - identificador do gestor, gerado pelo gestor principal, através do método *UUID.randomUUID* da biblioteca Java.

2. **host_address** - valor JSON com o endereço do gestor, com as propriedades: {username, publicDnsName, publicIpAddress, privateIpAddress, coordinates, region, place}.

3. **kafka_bootstrap_servers** - lista dos endereços onde estão a executar agentes kafka.

No inicio da execução de um gestor local, é iniciado o *docker swarm* com um nó do tipo *manager*, sendo esse nó o próprio *host* onde está a executar. São também iniciados cinco ciclos periódicos, dois para sincronizar os valores dos nós e contentores guardados na base de dados com o estado do *docker swarm*, outros dois ciclos para monitorizar os nós e contentores geridos pelo gestor local, e outro ciclo para enviar *heartbeats* ao gestor principal.

O gestor local inicia ainda um servidor HTTP disponibilizando um API visível na tabela 4.2. O API permite que o Hub de gestão comunique diretamente com um gestor local, para interagir com os contentores e nós do *docker swarm* que controla.

Base de dados

Os dados de um gestor local são guardados numa base de dados H2 em memória, com o esquema baseado na biblioteca da base de dados indicado anteriormente na secção 4.2.1. Os valores iniciais são obtidos através do Kafka, como explicado mais à frente, na secção de sincronização de dados 4.2.3.

Ciclo de sincronização da base de dados com o *docker swarm*

Tal como no gestor principal, a sincronização dos nós e contentores da base de dados com o *docker swarm* é feita através de um ciclo periódico que executa de 10 em 10 segundos.

Ao guardar os dados na base de dados, é possível detetar alterações do estado do *docker swarm*, e comunicá-las ao gestor principal.

A implementação da sincronização é semelhante ao que foi feito no gestor principal, explicada em mais detalhe na secção 4.2.3, mas sem a parte referente aos *heartbeats*.

Monitorização de nós e contentores

A monitorização de nós e contentores foi definida no trabalho anterior, através de dois ciclos periódicos com monitorização, análise, planeio e execução. Neste trabalho, o foco foi na parte da execução, com a aplicação de algoritmos que procuram otimizar a localização escolhida de onde iniciar os nós, ou replicar/migrar os contentores.

O ciclo de monitorização de nós inclui:

1. obtém as métricas do *prometheus* a executar em cada nó;
2. aplica as métricas simuladas atribuídas a cada nó;

Tabela 4.2: API de um gestor local. *Endpoints* relativos a /api.

Pedido	Endpoint	Descrição
POST	/apps/{app}/launch	Inicia todos os serviços da aplicação {app}, dentro de contentores.
GET	/containers	Obtém a lista de contentores.
GET	/containers/{id}	Obtém o contentor com o id `{id}`.
POST	/containers	Lança um contentor num endereço, ou numa coordenada, passado no corpo do pedido.
DELETE	/containers/{id}	Pára o contentor com id `{id}` no *docker swarm*.
POST	/containers/{id}/replicate	Replica o contentor com id `{id}`, para o endereço passado no corpo do pedido.
POST	/containers/{id}/migrate	Migra o contentor com id `{id}`, para o endereço passado no corpo do pedido.
GET	/containers/{id}/logs	Obtém os registos do contentor com id `{id}`.
POST	/containers/sync	Sincroniza os contentores na base de dados com o *docker swarm*.
POST	/hosts/cloud	Inicia e configura uma instância virtual na *cloud*.
GET	/nodes	Obtém a lista de nós.
GET	/nodes/{id}	Obtém o nó com o id `{id}`.
POST	/nodes	Adiciona um nó no endereço, ou perto de uma coordenada, passado no corpo do pedido.
PUT	/nodes/{id}	Atualiza as propriedades do nó com id `{id}`, com os valores passados no corpo do pedido.
DELETE	/nodes/{id}	Remove o nó com id `{id}` do *docker swarm*.
PUT	/nodes/{id}/join	Re-adiciona o nó com id `{id}` ao *docker swarm*.
DELETE	/nodes/{publicIp}/{privateIp}	Remove o nó do *host* com IP público {publicIp} e IP privado {privateIp}, do *docker swarm*.
POST	/nodes/sync	Sincroniza os nós na base de dados com o *docker swarm*.

3. aplica as regras definidas a cada nó, baseado no *template* drools visível na listagem 4.4.
4. decide qual a ação a tomar baseado nas decisões de cada nó, que foram baseadas nas métricas recolhidas e regras definidas. As ações possíveis são: parar um dos nós, iniciar outro nó, ou não fazer nada.

5. caso seja preciso iniciar um nó, aplica um algoritmo de seleção de localização.

Após o ciclo de monitorização de nós ser executado, se se decidir que é preciso iniciar um nó para remover a carga de um nó sobrecarregado, é escolhido um *host* o mais próximo possível desse nó (baseado em coordenadas), no qual será iniciado o novo nó e migrado para lá um contentor aleatório presente no nó sobrecarregado.

Listagem 4.4: *Template drools* para definir regras aplicadas a nós.

```
template header
rule
ruleId
eventType
decision
priority
package pt.unl.fct.miei.usmanagement.manager.management.rulesystem;
import pt.unl.fct.miei.usmanagement.manager.services.rulesystem.decision.Decision;
global pt.unl.fct.miei.usmanagement.manager.services.rulesystem.decision.Decision
    ↪ hostDecision;
template "host"
rule "rule_host_@{ruleId}" salience @{priority}
when
   @{eventType}(@{rule})
then
   hostDecision.setDecision(@{decision});
   hostDecision.setPriority(@{priority});
end
end template
```

O ciclo de monitorização de contentores inclui:

1. obtém as métricas relativas a cada contentor, através das métricas disponibilizadas pelo *docker*;
2. aplica as métricas simuladas atribuídas a cada aplicação, serviço e contentor, por esta ordem;
3. aplica as regras definidas a cada aplicação, serviço e contentor, por esta ordem, baseado num *template* drools semelhante à listagem 4.4.
4. decide qual a ação a tomar baseado nas decisões de cada nó: replicar um contentor, migrar um contentor, ou não fazer nada;
5. caso seja preciso migrar ou replicar um contentor, aplica o algoritmo de seleção de localização.

Após o ciclo de monitorização de contentores ser executado, e se decida que é preciso migrar ou replicar um contentor, é obtida informação sobre o número de acessos aos serviços associada à localização, através do monitor de pedidos presente em cada nó. Depois é calculado o ponto geográfico ótimo (ponto médio com peso) onde o contentor devia estar, através do algoritmo 1. De seguida procura-se um nó o mais próximo possível desse ponto geográfico, que tenha capacidade para executar o contentor. Está definido

que um nó com percentagem de ocupação de RAM e CPU menor a 90%, incluindo o RAM definido que o serviço necessita, esteja disponível para iniciar o contentor.

Para iniciar um contentor, o gestor local executa o algoritmo 2, por forma a escolher uma porta pública para o serviço, e evitar lançar contentores em portas já ocupadas.

Heartbeat

O gestor local inicia um ciclo periódico para enviar um *heartbeat* ao gestor principal, por defeito a cada 15 segundos. Como o gestor local desconhece a existência do gestor principal, o *heartbeat* é enviado para um tópico kafka, que depois é consumido pelo gestor principal.

4.2.2.3 Hub de gestão

A aplicação *web* desenvolvida no trabalho anterior foi alterada e completada para abranger todos os aspetos possíveis dos gestores. Continua a usar React [17] como *framework*, mas passou agora também a usar Typescript [18] para tornar o código tipificado, e portanto mais seguro [2]. Foi usado Redux [19] para a gestão dos dados da aplicação, o que permite centralizar todos os dados provenientes do gestor principal e gestores locais, e ter acesso fácil aos dados em qualquer componente da aplicação.

Características não funcionais

As características comuns a uma aplicação *web* não foram esquecidas. Foi implementado autenticação usando *basic authentication* [20], com uma página de autenticação, visível na figura 4.1, para autenticar utilizadores. A aplicação inclui *feedback* de ações e erros ao utilizador com recurso a barras de progresso e mensagens de sucesso ou erro. Contém prevenção de ações incorretas com o uso de javascript e CSS para esconder e/ou desativar componentes. Foi incluído também um esquema de cores claro e escuro, filtros de dados através de uma barra de pesquisa, listas paginadas para evitar mostrar demasiada informação na mesma página, e adaptações ao tamanho da janela escondendo e mostrando dinamicamente elementos consoante o tamanho da janela.

Foram desenvolvidos componentes genéricos, típico do *framework* React, para facilitar a extensão da aplicação no futuro. Entre os componentes estão listas genéricas paginadas e filtradas, formulários com validação de dados, botões, modais, *tabs*, menus de contexto (exemplo na figura 4.2) e mapas geográficos.

O código da aplicação foi escrito a pensar na extensibilidade por parte das outras duas vertentes do sistema abordadas em outras dissertações, a monitorização e a gestão de dados. O objetivo é que o Hub de gestão seja usado no futuro como consola de gestão do

[17] React: https://reactjs.org
[18] Typescript: https://www.typescriptlang.org
[19] Redux: https://redux.js.org
[20] Especificação da autenticação básica: https://tools.ietf.org/html/rfc7617

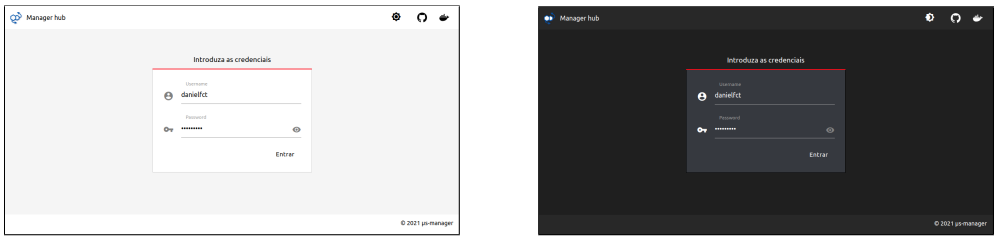
Algoritmo 1 Cálculo dos pontos médios de acesso aos serviços

```
servicesWeight ← GetServicesWeight()
for all (service, weights) ∈ servicesWeight do
    middlePoint ← GetWeightedMiddlePoint(weights)
    middlePoints[service] ← middlePoint
end for
return middlePoints

função GetServicesWeight
    nodes ← db.nodes[ready]
    for all n ∈ nodes do
        requests ← http.get(http://${n}:1919/api/location/requests?aggregation)
        for all (service, count) ∈ requests.serviceRequests do
            weights[service] ← weights[service] ∪ (n, count)
        end for
    end for
    return weights
end função

função GetWeightedMiddlePoint(weightedPoints)
    totalWeight ← Σ_{n=0}^{weightedPoints.size−1} weightedPoints[n].weight
    x ← 0, y ← 0, z ← 0
    for all (p ∈ weightedPoints) do
        weight ← p.weight
        lat ← p.coordinates.latitude
        lon ← p.coordinates.longitude
        lat_rad ← lat × π ÷ 180          ▷ Converter a latitude e longitude em radianos
        lon_rad ← lon × π ÷ 180
        x_n ← cos lat_rad × cos lon_rad  ▷ Converter radianos em coordenadas cartesianas
        y_n ← cos lat_rad × sin lon_rad
        z_n ← sin lat_rad
        x ← x + x_n × weight             ▷ Somar o peso deste ponto
        y ← y + y_n × weight
        z ← z + z_n × weight
    end for
    x ← x ÷ totalWeight
    y ← y ÷ totalWeight
    z ← z ÷ totalWeight
    hypersphere ← √(x² + y²)   ▷ Converter as coordenadas cartesianas em latitude e
longitude
    lat_rad ← atan2 z, hypersphere
    lon_rad ← atan2 y, x
    lat ← lat_rad × 180 ÷ π
    lon ← lon_rad × 180 ÷ π
    return (lat, lon)
end função
```

Algoritmo 2 Escolha da porta exterior de um serviço s num nó n

```
1: command ← ss -lntu | grep LISTEN | awk '{print $5}' | rev | cut -d: -f1 | rev
2: usedPorts ← SSH(n.hostname, command)
3: port ← s.port
4: while true do
5:     if port ∉ usedPorts then
6:         return port
7:     end if
8:     port ← port + 1
9: end while
10:
```

sistema, para gerir todas as vertentes do sistema, e não só a vertente que esta dissertação aborda.



(a) Tema claro. (b) Tema escuro.

Figura 4.1: Página de autenticação do Hub de gestão.

(a) Tema claro. (b) Tema escuro.

Figura 4.2: Menu de contexto associado a um contentor.

Características funcionais

Para além das características não funcionais, a aplicação implementa um conjunto de funcionalidades para permitir visualizar o progresso do sistema e interagir manualmente com o mesmo.

Comunica com o gestor principal através de HTTP [21], com troca de mensagens em formato JSON [22], para obter, adicionar, alterar e/ou remover entidades do sistema, como aplicações, serviços, condições, regras e métricas simuladas, bem como iniciar contentores, instâncias virtuais na *cloud*, nós, e lançar os componentes de apoio.

Comunica com os gestores locais, também através de HTTP, com troca de mensagens em formato JSON, para executar operações diretamente nos gestores locais, como alterar o estado de contentores e dos nós.

A aplicação está divida por páginas, geridas através de um *react-router* [23], que implementam todas as funcionalidades do hub de gestão:

- **Página principal**. É composta por um mapa mundo interativo, baseado na biblioteca *react-simple-maps* [24], com a informação, atualizada automaticamente a cada 15 segundos, sobre os contentores, nós e máquinas incluídas no sistema. Cada ponto no mapa representa um nó a executar no sistema, ou uma máquina gerida por um dos gestores. Ao passar o rato em cada ponto, é possível ver mais informação, nomeadamente, o endereço IP, os contentores a executar e a localização exata do nó.

- **Aplicações**. Na secção das aplicações, é apresentada uma lista das aplicações registadas no sistema. O ato de adicionar uma aplicação nova, é simplesmente preencher e submeter um formulário com todos os detalhes da aplicação. Ao clicar numa das aplicações, é mostrada outra página onde é possível editar os detalhes da aplicação escolhida, associar serviços e alterar a prioridade de cada um, associar regras à aplicação, ver as regras genéricas aplicadas a todas as aplicações, associar métricas simuladas à aplicação, e ver a lista das métricas simuladas aplicadas a todas as aplicações. Adicionalmente, é ainda possível iniciar todos os serviços da aplicação numa coordenada escolhida usando um mapa interativo, idêntico ao apresentado na página principal.

- **Serviços**. Na secção dos serviços, é apresentada a lista de serviços de todas as aplicações no sistema. Seguindo um dos serviços, é possível ver os detalhes do serviço, gerir as aplicações associadas, adicionar e remover dependências, ver a lista dos serviços dependentes, gerir as previsões, adicionar e remover regras, ver as regras genéricas aplicadas a todos os serviços, adicionar e remover métricas simuladas, e ver a lista das métricas simuladas genéricas.

- **Contentores**. Em relação aos contentores, é apresentada a lista de todos os contentores reconhecidos pelo gestor principal. Através do botão presente na parte superior direita da página, é possível sincronizar os contentores na base de dados do gestor principal, com todos os *docker swarms* dos gestores. O pedido é enviado

[21] HTTP restful: https://restfulapi.net
[22] JSON: https://www.json.org
[23] React-router: https://reactrouter.com
[24] React simple maps: https://www.react-simple-maps.io

para o gestor principal, que depois envia um pedido a cada um dos gestores locais, para sincronizar a sua própria base de dados e enviar os dados atualizados sobre os seus contentores. Existem duas possibilidades para lançar um contentor: por localização e numa máquina especifica. A localização é escolhida através de um mapa interativo, calculando automaticamente as coordenadas. O pedido de iniciação do contentor é depois enviado para o gestor principal, que determina qual dos gestores locais deve executar a ação. Ao clicar num dos contentores é possível ver todos os detalhes do contentor, incluindo as portas, *labels* associadas e os seus registos. E ainda gerir as regras associadas, ver a lista das regras genéricas, gerir as métricas simuladas do contentor, e ver a lista das métricas simuladas genéricas aplicadas a todos os contentores. Está também definido um botão associado a um *link* que abre um separador com o URL do serviço do contentor. Adicionalmente, e apenas nos contentores de serviços, é possível migrar e replicar o contentor para um nó escolhido no momento.

- **Hosts**. A secção dos *hosts* está dividida em instâncias virtuais na *cloud* e máquinas *edge*. Adicionar uma nova instância *cloud* é feito através da seleção das coordenadas, num mapa interativo que inclui a localização de todas as zonas suportadas. O pedido é enviado para um gestor local ou para o gestor principal, que escolhe a zona AWS suportada que se encontra mais perto. Registar uma máquina *edge* é simplesmente preencher e submeter toda a informação necessária, num formulário. Ao clicar num dos *hosts* é possível ver os detalhes do *host*, gerir as regras associadas, ver a lista das regras genéricas aplicadas aos *hosts*, associar métricas simuladas, e ver a lista das métricas simuladas genéricas. Existem ainda duas secções, uma para executar comandos SSH e outra para carregar ficheiros no *host*, através de SFTP.

- **Nós**. Na página dos nós, é apresentada a lista de todos os nós reconhecidos pelo gestor principal. No canto superior direito, existe um botão para sincronizar os nós na base de dados do gestor principal com os *docker swarms* de todos os gestores, semelhante ao que é feito para os contentores. O pedido é enviado para o gestor principal, que depois envia um pedido de sincronização de nós para cada gestor local. Ao clicar num dos nós são mostrados os seus detalhes, juntamente com as *labels* associadas ao nó. Em nós que se encontrem inseridos num *swarm*, é ainda possível executar uma operação para sair. E, caso contrário, se o nó não estiver num *swarm*, são apresentados dois botões, um para removê-lo definitivamente do sistema, outro para re-entrar no *swarm*.

- **Regiões**. Nesta secção são apresentadas as regiões suportadas pelo sistema. A funcionalidade de adicionar mais regiões não foi implementada, mas é algo que pode ser facilmente adicionado no futuro.

- **Regras**. A secção das regras está dividida em cinco partes: condições, regras aplicadas a *hosts*, regras aplicadas a aplicações, regras aplicadas a serviços e regras

aplicadas a contentores. Ao clicar numa das condições, é possível ver, editar, ou apagar a informação relativa à condição. Ao seguir uma das regras, é apresentada uma página para editar os detalhes da regra, bem como associar entidades à mesma. No caso das regras aplicadas a *hosts* é possível associar condições, instâncias *cloud*, e máquinas edge. Semelhantemente, nas regras aplicadas a aplicações, serviços, ou contentores, é possível associar a respetiva entidade a que são aplicadas. Nas regras aplicadas a *hosts* e serviços, é possível defini-la como sendo genérica, ou seja, é aplicada a todos os *hosts* ou serviços registados no sistema.

- **Métricas simuladas**. Parecido às regras, a página das métricas simuladas inclui uma secção para *hosts*, aplicações, serviços, e contentores. Após clicar numa das métricas simuladas, a página muda para ser apresentada a informação relativa à métrica, incluindo secções para a associar a entidades. No caso de métricas simuladas aplicadas a *hosts* é possível associar *cloud* e *edge hosts* aos quais é aplicada a métrica. Semelhantemente, no caso de métricas simuladas aplicadas a aplicações, serviços, e contentores, é possível associar as respetivas entidades. Tal e qual como nas regras, nas métricas simuladas de *hosts* e serviços, também é possível definir a regra como sendo genérica.

- **Balanceamento de carga**. A parte do balanceamento de carga inclui uma lista com os balanceadores de carga a executar em cada região. Clicando num dos balanceadores é possível aceder a mais detalhes sobre o mesmo.

- **Servidores de registo**. Sobre os servidores de registo, é apresentada uma lista com todos os servidores a executar em cada região. Ao aceder a um dos servidores de registo é apresentada uma página com os detalhes do servidor.

- **Gestores locais**. Secção para listar os gestores locais, bem como apresentar a informação de cada um. É possível ver ainda as listas dos nós e contentores associados a cada gestor.

- **Agentes Kafka**. Nesta secção estão os agentes kafka, com uma página para listar todos os agentes a executar nas diferentes regiões, e outra página para apresentar os detalhes de um agente após ser selecionado.

- **Secure shell**. Nesta secção da aplicação é possível executar comandos SSH e, através do protocolo SFTP, carregar ficheiros nos nós do sistema.

- **Registo**. A página dos registos inclui os *logs* do gestor principal. No canto superior direito, está localizado um botão para ativar o carregamento automático dos registos, que atualiza os registos de 15 em 15 segundos. E um elemento para selecionar o número de linhas apresentadas por página.

4.2.3 Sincronização de dados

A sincronização dos dados entre o gestor principal e os gestores locais é feita através de agentes kafka, como na figura 4.3. Em cada gestor, sempre que haja uma alteração numa tabela da base de dados de interesse, é enviado um registo serializado em formato JSON do Objeto de Transmissão de Dados (DTO) para o tópico kafka correspondente. E no caso de ser uma operação "*delete*", é enviada o valor `DELETE` na chave do registo, sendo o valor do registo o identificador da entidade a ser removida.

Para cada tipo de dados é criado um tópico kafka, sendo que um gestor local consome um total de 20 tópicos [25], e o gestor principal um total de 10 tópicos [26].

Para evitar o acoplamento das entidades na base de dados com o tipo de dados transmitidos, foi usado o gerador de código MapStruct [27], que permite gerar o mapeamento entre uma entidade JPA e um objeto DTO, controlado através de anotações Java e com deteção automática de propriedades. A implementação do mapeamento é gerada automaticamente na altura de compilação do programa, sendo que um exemplo de definição de um mapeamento pode ser visto na listagem 4.5. O *CycleAvoidingMappingContext* é uma classe definida para evitar ciclos infinitos, registando as entidades encontradas e reutilizando as suas referências quando são precisas novamente.

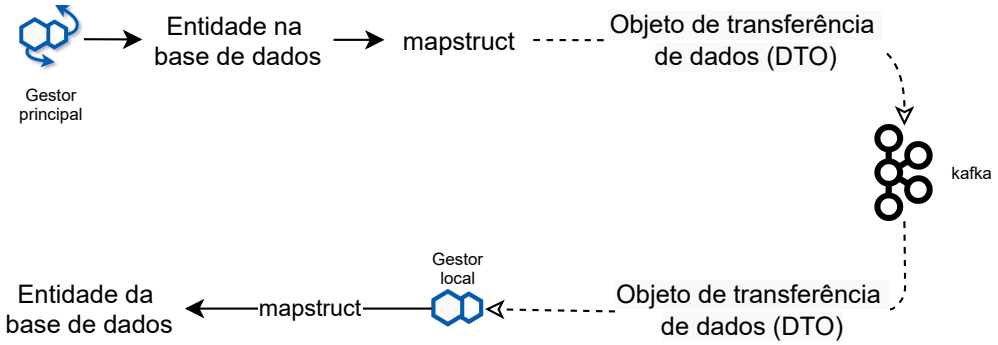


Figura 4.3: Sincronização de dados entre o gestor principal e um gestor local.

Listagem 4.5: Definição do mapeamento de uma entidade JPA para um DTO de um serviço, e vice-versa.

```
1 @Mapper(builder = @Builder(disableBuilder = true))
2 public interface ServiceMapper {
3
4   ServiceMapper MAPPER = Mappers.getMapper(ServiceMapper.class);
```

[25] Tópicos kafka consumidos por um gestor local: *apps*, *services*, *containers*,*cloud-hosts*, *edge-hosts*, *component-types*, *conditions*, *decisions*, *eips*, *fields*, *operators*, *simulated-host-metrics*, *simulated-app-metrics*, *simulated-service-metrics*, *simulated-container-metrics*, *host-rules*, *app-rules*, *service-rules*, *container-rules*, *value-modes*.

[26] Tópicos kafka consumidos pelo gestor principal: *containers*, *nodes*, *cloud-hosts*, *heartbeats*, *host-events*, *service-events*, *host-monitoring-logs*, *service-monitoring-logs*, *host-decisions*, *service-decisions*.

[27] Mapstruct: https://mapstruct.org

```
5
6   @IterableMapping(nullValueMappingStrategy = NullValueMappingStrategy.RETURN_DEFAULT)
7   Service toService(ServiceDTO serviceDTO, @Context CycleAvoidingMappingContext context);
8
9   @InheritInverseConfiguration
10  ServiceDTO fromService(Service service, @Context CycleAvoidingMappingContext context);
11
12 }
```

A base de dados de cada gestor é constantemente sincronizada para estar atualizada em relação aos dados reais. Em cada gestor, a informação entre o *docker swarm* que controla e a sua base de dados é sincronizada em dois ciclos que executam de 10 em 10 segundos, um para sincronizar os contentores, representado no algoritmo 3, outro para sincronizar os nós. Tanto um contentor como um nó é considerado dessincronizado se estiver presente no *docker swarm* e não na base de dados, se estiver na base de dados, mas não estiver presente no *docker swarm*, ou se pertencer a um gestor local e este não envia um *heartbeat* há mais de 60 segundos. Adicionalmente, um contentor é atualizado se os valores do endereço IP forem diferentes, e um nó é atualizado se os valores da disponibilidade, estatuto, estado, ou endereço IP forem diferentes.

No gestor principal, está definido ainda outro ciclo de sincronização, para sincronizar a informação na base de dados com as instâncias virtuais na AWS. O ciclo executa de 45 em 45 segundos, e embora os pedidos para as regiões suportadas sejam enviados em paralelo, demora alguns segundos até a sincronização ser executada na totalidade.

No inicio da configuração de uma entidade, quer sejam instâncias virtuais na *cloud*, contentores, ou nós, é inserida uma entrada numa tabela *configurations*, que fica associada à entidade através da sua identificação. No final da configuração, quer tenha sucesso ou não, a entrada é removida. Durante a sincronização de uma entidade (instâncias na *cloud*, contentores, nós), é verificado se existe alguma entrada nessa tabela, e caso se detete, a sincronização dessa entidade não é executada. Esse processo tem como objetivo evitar criar conflitos entre a sincronização e a execução normal do sistema. Devido ao facto de que a entidade só deve ser inserida na base de dados após ser totalmente configurada.

4.2.4 Componentes de apoio às operações

Os componentes de apoio são diversos serviços lançados dentro de contentores para suportar a execução do sistema. Fazem parte o proxy com autenticação básica 4.2.4.1 para proteger os dados expostos pelo servidor HTTP do docker. O servidor de registo 4.2.4.2 e o cliente de registo 4.2.4.3 para registo e descoberta de serviços. O monitor de pedidos 4.2.4.4 para obter informação sobre o número de pedidos a serviços externos por parte dos micro-serviços. O prometheus 4.2.4.7 para a obtenção de métricas dos nós do sistema. O balanceador de carga 4.2.4.5 para distribuir a carga dos pedidos feitos a serviços *front-end*. E o agente kafka 4.2.4.8 para sincronizar os dados entre os vários gestores.

Algoritmo 3 Sincronização entre os contentores da base de dados e do *docker swarm*.

```
1:  for all c ∈ swarm.containers do                          ▷ Adicionar contentores
2:      if c.id ∉ db.configurations then
3:          if c ∉ db.containers then
4:              db.containers ← db.containers ∪ {c}
5:          end if
6:      end if
7:  end for
8:  for all c ∈ db.containers do                    ▷ Remover e atualizar contentores
9:      if c.id ∉ db.configurations then
10:         if c ∉ swarm.containers then
11:             db.containers ← db.containers \ {c}
12:         else
13:             swarm_container ← swarm.containers[c.id]
14:             if c.address ≠ swarm_container.address then
15:                 c.address ← swarm_container.address
16:                 db.containers ← db.containers ∪ {c}
17:             end if
18:             if db.heartbeats[c.manager.id] + 60 < time.now then
19:                 c.state ← down
20:                 db.containers ← db.containers ∪ {c}
21:             end if
22:         end if
23:     end if
24: end for
25: for all m ∈ db.localManagers do          ▷ Atualizar o estado dos gestores locais
26:     if db.heartbeats[m.id] + 60 < time.now then
27:         m.state ← down
28:         db.localManagers ← db.localManagers ∪ {m}
29:     end if
30: end for
```

4.2.4.1 Proxy com autenticação básica

Este componente foi usado no trabalho anterior, e continua presente para garantir proteção aos dados dos *docker swarms* de cada gestor.

O API exposto pelo docker apenas está acessível localmente a cada *host*. Para ter acesso ao mesmo, é usado um Proxy NGINX [28], visível na figura 4.4, que executa em cada nó, e após autenticação com utilizador e palavra-passe, redireciona o pedido para a porta onde está a executar o servidor HTTP com o API do docker. Desta forma, os dados relativos ao docker são protegidos por autenticação básica. O proxy é lançado dentro de um contentor, no momento da configuração do nó.

[28] NGINX: https://www.nginx.com

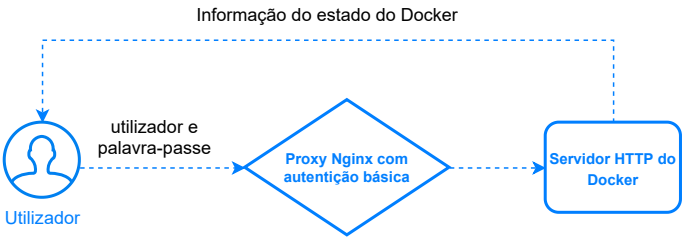


Figura 4.4: Proxy NGINX protegido com autenticação básica.

4.2.4.2 Servidor de registo

O servidor de registo integra o Eureka da Netflix [29] para permitir registar e descobrir serviços. A diferença para o componente usado no trabalho anterior, é que agora o servidor é executado dinamicamente, um em cada região, onde estão a executar contentores. O dinamismo é suportado pelo gestor principal, que deteta quando não existem contentores a executar e pára o servidor de registo 5 minutos depois. O atraso de 5 minutos tem a função de evitar que um servidor seja parado e reiniciado num curto espaço de tempo. Se entretanto for lançado um contentor na região, a ação de paragem é cancelada.

O servidor é executado num contentor e só pode ser lançado numa instância virtual na *cloud*, porque o endereço IP de todos os servidores de registo devem ser conhecidos no inicio da execução do sistema, para que consigam comunicar entre si.

Endereços IP elásticos

No inicio da execução do sistema, o gestor principal aloca um endereço IP elástico [30] da Amazon por região, como descrito anteriormente na secção 4.2.2.1. A lista dos endereços IP alocados é usada no servidor de registo para definir a localização de possíveis replicas de servidores. Desta forma, um servidor de registo tem conhecimento prévio de todos os outros possíveis servidores, possibilitando a replicação e sincronização dos serviços registados entre regiões distintas.

4.2.4.3 Cliente de registo e descoberta de serviços

É uma aplicação que foi estendida do trabalho anterior, implementada na linguagem Go[31], expondo um servidor HTTP, por defeito na porta 1906. Para além de ter sido adaptado para agora usar coordenadas no algoritmo de escolha da melhor instância, como no pseudocódigo do algoritmo 4, foram também introduzidos dois algoritmos novos, descritos na secção 4.2.4.3.

A aplicação executa no mesmo contentor do micro-serviço, expondo uma API, privada ao nível do contentor, com 6 *endpoints*, descritos na tabela 4.3.

[29] Eureka Netflix: https://github.com/Netflix/eureka
[30] Endereço IP elástico: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html
[31] Linguagem go: https://golang.org

Tabela 4.3: API do cliente de registo e descoberta de serviços. *Endpoints* relativos a /api.

Pedido	Endpoint	Descrição
POST	/register	Regista o serviço no servidor de registo.
GET	/services/{service}/endpoints	Obtém a instância mais próxima do serviço {service}.
GET	/services/{service}/endpoints?among=x	Obtém uma instância aleatória para o serviço {service} entre as x instâncias mais perto.
GET	/services/{service}/endpoints?range=d	Obtém uma instância aleatória para o serviço {service} começando a procura à distância de d quilómetros, e duplicando em cada iteração.
GET	/services/{service}/endpoints	Obtém todas as instâncias registadas em nome do serviço {service}.
POST	/metrics	Adiciona uma nova monitorização desta instância. O corpo do pedido é composto por: {service, latitude, longitude, count}.

As funcionalidades de registo automático, *heartbeat*, e descoberta de serviços, desenvolvidas no trabalho anterior, continuam presentes.

Quanto ao registo automático, caso o argumento `register` passado à aplicação tiver o valor `true`, o micro-serviço é registado automáticamente no servidor de registo um certo tempo depois de iniciar, sendo 30 segundos por omissão. Se o micro-serviço implementar o registo manual da sua instância (por exemplo, como na listagem 4.7), o registo automático é cancelado, caso esteja ativo e ainda não tenha executado.

Para implementar o *heartbeat*, a aplicação verifica periodicamente, num período de tempo de 30 segundos, se o micro-serviço ainda está a executar, através de uma procura na lista de processos em execução no contentor. Caso se verifique, é enviada uma mensagem *heartbeat* para o servidor de registo, para este reconhecer a vitalidade do micro-serviço em questão. Caso o processo não seja encontrado durante três ciclos consecutivos, a instância do micro-serviço é revogada no servidor de registo, e o contentor é terminado.

Visto que a funcionalidade de descoberta de serviços não é usada para descobrir serviços *frontend*, porque é usado o balanceador de carga, a monitorização é enviada pelo próprio serviço, após este receber um pedido por parte de um utilizador, como descrito na listagem 4.6. A localização do utilizador é obtida através dos cabeçalhos X-Latitude e X-Longitude, que foram colocados no pedido pelo balanceador de carga NGINX, como explicado na sua secção 4.2.4.5.

Listagem 4.6: Exemplo do envio de um registo para o monitor de pedidos, por parte de um serviço *frontend*.

```
1 const url = "http://localhost:1906/api/metrics"
2 func sendMonitoringData(r *http.Request) {
3   latitude := r.Header.Get("X-Latitude")
4   longitude := r.Header.Get("X-Longitude")
```

```
5   var jsonStr = []byte(fmt.Sprintf(`{
6     "service": %s,
7     "latitude": %s,
8     "longitude": %s,
9     "count": 1
10  }`, serviceName, latitude, longitude))
11  req, _ := http.NewRequest("POST", url, bytes.NewBuffer(jsonStr))
12  req.Header.Set("Content-Type", "application/json")
13  client := &http.Client{}
14  resp, _ := client.Do(req)
15  defer resp.Body.Close()
16 }
```

Para suportar a descoberta de serviços, o cliente de registo e descoberta de serviços primeiro examina a sua *cache* para ver se existe alguma instância do serviço em questão. A *cache* de uma instância de um serviço fica ativa durante 10 segundos, para evitar constantes pedidos ao servidor de registo. Se a cache estiver vazia, ou o tempo tiver expirado, o cliente acede ao servidor de registo 4.2.4.2 para obter a lista de instâncias registados do serviço. Depois, entre a lista, é executado um algoritmo para escolher a melhor instância, como descrito na próxima secção 4.2.4.3.

Algoritmos de descoberta de serviços

Foram feitas alterações na funcionalidade de descoberta de serviços, notoriamente, no algoritmo de seleção da melhor instância, que agora passa a usar a latitude e a longitude para selecionar a melhor instância para o serviço pretendido. Com a intenção de não sobrecarregar demasiado uma instância com proximidade a muitos serviços dependentes, foram introduzidos dois algoritmos não determinísticos, baseados em aleatoriedade entre um certo número de instâncias.

O algoritmo para calcular a distância é o mesmo usado nos gestores. Usando a informação da latitude e longitude associado a cada instância, e com o uso da fórmula de Haversine, é trivial calcular a distância:

Algoritmo 4 Cálculo de distância entre duas instâncias de serviços.

1: $latitude_1 \leftarrow servidor_1.latitude$
2: $longitude_1 \leftarrow servidor_1.longitude$
3: $latitude_2 \leftarrow servidor_2.latitude$
4: $longitude_2 \leftarrow servidor_2.longitude$
5: $\phi_1 \leftarrow latitude_1 * \pi/180$
6: $\phi_2 \leftarrow latitude_2 * \pi/180$
7: $\delta_y \leftarrow (longitude_2 - longitude_1) * \pi/180$
8: $r_e \leftarrow 6371e3$
9: **return** $\arccos(\sin\phi_1 * \sin\phi_2 + \cos\phi_1 * \cos\phi_2 * \cos\delta_y) * r_e$

As distâncias são depois usadas para encontrar a instância que cumpre os requisitos do argumento passado pelo *query* (*amoung* ou *range*) do URL. Primeiro, a lista é ordenada

pela distância. Depois, no caso de ser pedido uma instância aleatória entre as x instâncias mais perto (...?amoung=x), basta apenas escolher um aleatoriamente entre os x primeiros elementos da lista. No caso de ser pedido uma instância aleatória começando a procura à distância de d quilómetros (...?range=d), é feita uma iteração na lista, efetuando-se uma procura aleatória entre as instâncias à distância de d quilómetros. Se nenhuma instância for encontrada, continua-se a iteração, é feita outra procura aleatória com as instâncias até ao dobro da distância, e assim sucessivamente.

APIs do cliente de registo e descoberta de serviços

Para disponibilizar acesso fácil ao API do cliente de registo e descoberta de serviços, com recurso a uma especificação *OpenAPI* [32] e ao gerador de código *Swagger codegen* [33], foram gerados APIs de clientes para quatro linguagens diferentes: C++, C, Python e Node, para além das linguagens já suportadas no trabalho anterior: Java e Go.

As APIs são incluídas nos micro-serviços implementados na respetiva linguagem, o que possibilita que executem facilmente chamadas REST ao API, para efetuarem um registo manual, ou fazerem a descoberta de um serviço externo.

As APIs estão disponíveis através do GitHub, nos repositórios Java, Go, C++, C, Python, e Node, que depois são descarregados, instaladas na imagem Docker e importadas no código dos micro-serviços.

Um exemplo de registo de um micro-serviço implementado em C++, pode ser observado na listagem 4.7. O API disponível nas outras linguagens é o mesmo, apenas é diferente o procedimento para as execuções das chamadas aos *endpoints* do API do cliente de registo e descoberta de serviços.

Listagem 4.7: Registo de um micro-serviço implementado na linguagem C++.

```
#include "api/EndpointsApi.h"
using namespace pt::unl::fct::miei::usmanagement::manager::client::api;

void register()
{
    std::shared_ptr<ApiClient> apiClient(new ApiClient);
    std::shared_ptr<ApiConfiguration> apiConfig(new ApiConfiguration);
    apiConfig->setBaseUrl("http://localhost:1906/api");
    apiClient->setConfiguration(apiConfig);
    EndpointsApi api(apiClient);
    api.registerEndpoint().wait();
}
```

Para a obtenção de uma instância de um serviço externo, basta chamar o API do cliente de registo e descoberta de serviços. Um exemplo, na linguagem Go, pode ser visto na listagem 4.8.

[32] OpenAPI: https://www.openapis.org
[33] Swagger codegen: https://swagger.io/tools/swagger-codegen

Listagem 4.8: Obtenção de um *endpoint* de um serviço externo, num micro-serviço implementado em Go.

```
func getEndpoint(service string) (*registration.Endpoint, error) {
  apiClient := registration.NewAPIClient(registration.NewConfiguration())
  ctx := context.Background()
  endpoint, _, err := apiClient.EndpointsApi.GetServiceEndpoint(ctx, service)
  if err != nil {
    return nil, err
  }
  return &endpoint, nil
}
```

Um exemplo de um resultado da descoberta está disponível na listagem 4.9. Contém o valor do identificador da instância, que é composto pelo SHA1 [34] da concatenção do nome com o *endpoint* da instância. E contém o valor do *endpoint* da instância escolhida para o serviço externo.

Listagem 4.9: Exemplo do resultado da chamada do API para a descoberta de serviços.

```
{
    "instanceId": "f8fbf50b21357cab6ea73ed918db2934de9740f3",
    "endpoint": "3.122.248.143:5000"
}
```

4.2.4.4 Monitor de pedidos

O monitor de pedidos é uma aplicação desenvolvida no trabalho prévio 3.1, desenvolvida em Go, que executa dentro do seu próprio contentor em cada nó do sistema. As alterações efetuadas foram ao nível do tipo de dados monitorizados, que passou de cidade/pais/-continente para coordenadas (latitude/longitude). E o API e respetivas respostas foram modificadas para estarem em conformidade com a arquitetura REST.

No inicio da sua execução, inicia um servidor HTTP expondo uma API descrita na tabela 4.4.

O pedido para adicionar uma monitorização é enviada periodicamente, como na listagem 4.18, pelo cliente de registo e descoberta de serviços, que através de um POST para o endpoint */api/location/requests*, envia no corpo da mensagem os detalhes relativos à descoberta de serviços que efetuou.

Os dados guardados no monitor de pedidos, são periodicamente requeridos pelo gestor local que controla o nó. Esses dados são depois usados no algoritmo 1, para selecionar a melhor localização para onde migrar ou replicar contentores.

[34] Especificação de SHA1: https://www.ietf.org/rfc/rfc3174.txt

Tabela 4.4: API do monitor de pedidos. *Endpoints* relativos a /api.

Pedido	Endpoint	Descrição
GET	/location/requests	Lista toda a monitorização registada.
GET	/location/requests? aggregation	Lista toda a monitorização registada agregada por serviço, nos últimos 60 segundos.
GET	/location/requests? aggregation&interval={ms}	Lista toda a monitorização registada agregada por serviço, nos últimos {ms} milisegundos.
POST	/location/requests	Adiciona uma nova monitorização. O corpo do pedido é composto por: {service, latitude, longitude, count}.

4.2.4.5 Balanceador de carga

Este componente contém os ficheiros necessários para iniciar um balanceador de carga, baseado no NGINX, configurado com o módulo Ngx http GeoIP2 [35]. A configuração NGINX é baseada no template 4.13, que é gerido pelo API de configuração, descrito na próxima secção 4.2.4.6. O módulo GeoIP2 é carregado dinamicamente pelo servidor NGINX, ficando disponível a informação sobre a localização dos pedidos usando a base de dados GeoIP2. Neste caso, apenas é usada a latitude e longitude da origem do pedido, mas estão disponíveis outros valores que são ignorados, como o continente, o pais, a cidade e o código postal.

O balanceador de carga é lançado dentro de um contentor, sendo que a configuração do GeoIP2 é feita na altura da construção da imagem docker, antes de ser instalado o NGINX na imagem. Nesse momento, é executado um *script*, visível na listagem 4.10, para obter a base de dados GeoLite2-City. Sendo que é necessário que exista o ficheiro de configuração 4.11, para ser garantido o acesso à base de dados.

Listagem 4.10: *Script* para configuração da base de dados GeoIP2, executado na criação da imagem docker do balanceador de carga.

```
1  ARG GEOIPUPDATE_VERSION=4.5.0
2  COPY geoip/GeoIP.conf /usr/local/etc/
3  RUN wget
4        https://github.com/maxmind/geoipupdate/releases/download/v${GEOIPUPDATE_VERSION}/
5        geoipupdate_${GEOIPUPDATE_VERSION}_linux_amd64.tar.gz && \
6     tar -xf geoipupdate_${GEOIPUPDATE_VERSION}_linux_amd64.tar.gz && \
7     cp geoipupdate_${GEOIPUPDATE_VERSION}_linux_amd64/geoipupdate /usr/local/bin && \
8     rm -r geoipupdate_${GEOIPUPDATE_VERSION}_linux_amd64 && \
9     mkdir /usr/local/share/GeoIP && \
10    geoipupdate
```

[35] NGINX: https://www.nginx.com/products/nginx/modules/geoip2

Listagem 4.11: Ficheiro de configuração do GeoIP2, usado para obter a base de dados GeoLite2-City.

```
AccountID ...
LicenseKey ...
EditionIDs GeoLite2-City
```

Um balanceador de carga pode ser iniciado com servidores configurados de inicio, passando um valor JSON, do género apresentado na listagem 4.12, na variável de ambiente *SERVERS* do contentor. O gestor do balanceador de carga 4.2.4.6, responsável pela gestão dos servidores registados, analisa o valor dessa variável, e inicia o ficheiro de configuração NGINX com os servidores indicados.

Listagem 4.12: Exemplo do valor JSON que pode ser passado na inicialização de um balanceador de carga.

```
{
    "service": "sock-shop-frontend",
    "servers": [
        {
            "server": "202.193.203.125:5000",
            "latitude": 39.575097,
            "longitude": -8.909794,
            "region": "EUROPE"
        }
    ]
}
```

4.2.4.6 Gestor do balanceador de carga

Este componente, adaptado do trabalho anterior, é uma aplicação desenvolvida em Go, e iniciada no mesmo contentor do servidor NGINX, que inicia um servidor HTTP, disponibilizando um API, descrita na tabela 4.5, para gerir os servidores registados num balanceador de carga NGINX.

Em relação ao trabalho desenvolvido anteriormente, o componente passou a suportar servidores de diversos serviços, tendo sofrido alterações ao nível do API, e o tipo de estruturas de dados onde é guardada a informação sobre os servidores dos serviços, passando de uma lista para um mapa de listas.

Sempre que um servidor é adicionado ou removido por este componente, através do seu API, é usado o template 4.13 para gerar o ficheiro de configuração NGINX, que depois é carregado dinamicamente pelo servidor NGINX, efetivamente alterando os servidores disponíveis para o balanceamento dos pedidos.

Listagem 4.13: *Template* usado para gerar o ficheiro de configuração NGINX dinamicamente.

```
load_module "modules/ngx_http_geoip2_module.so";
user nginx;
```

Tabela 4.5: API para configuração dos servidores registados no balanceador de carga. Endpoints relativos a /api.

Pedido	Endpoint	Descrição
GET	/servers	Obtém os servidores de todos os serviços registados neste balanceador de carga.
GET	/{service}/servers	Lista todos os servidores do serviço `{service}` registados neste balanceador de carga.
POST	/{service}/servers	Adiciona servidores novos ao serviço `{service}`. Pedido: `[{server, latitude, longitude, region}]`.
DELETE	/{service}/servers/{server}	Remove o servidor server do serviço `{service}`.

```
worker_processes auto;
events {
  worker_connections 1024;
}
http {
    geoip2 /usr/local/share/GeoIP/GeoLite2-City.mmdb {
        $geoip2_location_latitude default=-1 location latitude;
        $geoip2_location_longitude default=-1 location longitude;
    }
    {{range $service, $servers := .}}
    upstream {{$service}} {
        least_conn; {{range $servers}}
        server {{.Server}}; #{{.Latitude}} {{.Longitude}} {{.Region}}{{end}}
    }
    {{end}}
    server {
        listen 80;
        server_name load-balancer.com;
        include /etc/nginx/conf.d/*.conf;
        {{range $service, $servers := .}}
        location /{{$service}}/ {
            proxy_pass http://{{$service}}/;
            proxy_set_header X-Latitude $geoip2_location_latitude;
            proxy_set_header X-Longitude $geoip2_location_longitude;
        }
        {{end}}
        location /_/api/ {
            auth_basic "Restricted";
            auth_basic_user_file /etc/nginx/.htpasswd;
            proxy_pass http://localhost:1906/api/;
            proxy_set_header X-Forwarded-Host $host;
            proxy_set_header Authorization "";
            proxy_redirect off;
            proxy_set_header X-Latitude $geoip2_location_latitude;
```

```
37            proxy_set_header X-Longitude $geoip2_location_longitude;
38        }
39        location = /404.html {
40            internal;
41        }
42    }
43 }
```

4.2.4.7 Prometheus e Node exporter

O Prometheus é um sistema *open-source*, que foi usado como componente de monitorização de nós no sistema, desenvolvido no trabalho anterior, e foi novamente incluído no sistema sem ter sofrido quaisquer modificações. As modificações feitas foram ao nível das *queries* definidas na tabela 4.6, onde foi adicionada a *query* FILESYSTEM_AVAILABLE_SPACE, usada para verificar se um nó tem capacidade no disco para transferir uma eventual imagem docker de um contentor, que deveria executar.

O componente executa dentro de um contentor docker, em todos os nós do sistema. Juntamente com o Node exporter, permite obter métricas sobre o sistema operativo *NIX e o *hardware* da máquina onde executa.

O Node exporter é configurado automaticamente em cada nó, na fase de configuração do mesmo, após este entrar no *docker swarm* de um gestor. Como o Node exporter executa num processo em *background*, foi preciso desenvolver código para executar comandos SSH e *bash* assíncronos, que executam continuamente em *background*. Para executar o processo em *background*, foi usado o comando *nohup* [36], que ordena ao sistema operativo para continuar a executar o processo, mesmo após a sessão SSH ou *bash* ser terminada.

Estão definidos um total de 7 *queries* feitas ao Prometheus para obter as métricas das máquinas dos nós, apresentadas na tabela 4.6.

4.2.4.8 Agentes Kafka

O kafka [37] é uma plataforma distribuída de *streaming* de eventos. É um componente lançado em cada região, dentro de um contentor, usada para a sincronização dos dados entre os gestores.

Tanto no gestor principal, como num gestor local, é usado o suporte kafka proporcionado pelo spring boot [38], com a seguinte configuração:

Listagem 4.14: Configuração do kafka nos gestores.

```
1 // Configuração do produtor Kafka
2 public Map<String, Object> producerConfigs() {
3     Map<String, Object> props = new HashMap<>();
4     props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
```

[36] Comando linux nohup: https://linux.die.net/man/1/nohup
[37] Apache kafka: https://kafka.apache.org
[38] Spring boot: https://spring.io/projects/spring-boot

Tabela 4.6: *Queries* definidas e usadas para obter as métricas ao Prometheus.

Nome	Query
AVAILABLE_MEMORY	node_memory_MemAvailable_bytes
TOTAL_MEMORY	node_memory_MemTotal_bytes
MEMORY_USAGE	node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes
AVAILABLE_MEMORY_RATE	node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes
MEMORY_USAGE_PERCENTAGE	100 * (1 - ((avg_over_time(node_memory_MemFree_bytes[5m]) + avg_over_time(node_memory_Cached_bytes[5m]) + avg_over_time(node_memory_Buffers_bytes[5m])) / avg_over_time(node_memory_MemTotal_bytes[5m]))))
CPU_USAGE_PERCENTAGE	100-(avg by (instance) (irate(node_cpu_seconds_total{ job=node_exporter,mode=idle}[5m]))*100)
FILESYSTEM_AVAILABLE_SPACE	node_filesystem_avail_bytes{mountpoint='/'}
NETWORK_BYTES_RECEIVED	rate(node_network_receive_bytes_total[5m])

```
5       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
6       return props;
7   }
8   // Configuração do consumidor Kafka
9   public Map<String, Object> consumerConfigs() {
10      Map<String, Object> props = new HashMap<>();
11      props.put(ConsumerConfig.GROUP_ID_CONFIG, UUID.randomUUID().toString());
12      props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
13      props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
14      props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
15      props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
16      return props;
17  }
```

Num registo kafka, o produtor define a chave como sendo um JSON serializado a partir de um objeto Java contendo a propriedade *managerId*, e a propriedade *operation*. *managerId* contém o identificador do gestor que criou o registo. *operation* contém um valor optional DELETE para indicar que o registo tem o objetivo de apagar uma entidade da base de dados, sendo o valor do registo o identificador da entidade em questão. Caso o valor da *operation* seja omitido, assume-me que o registo contém um valor JSON a representar uma entidade, que eventualmente será inserida ou atualizada na base de dados do consumidor.

Após o lançamento do primeiro agente kafka, o gestor principal tem a função de enviar todos os dados da base de dados relevantes para os tópicos kafka correspondentes. Como foi descrito na secção da sincronização de dados 4.2.3, o gestor local processa um total de 20 tópicos, e o gestor principal um total de 10 tópicos. Em cada um dos gestores, após quaisquer alterações aos dados na base de dados, relevantes aos outros gestores, é

produzido um novo registo no tópico kafka correspondente, que é consumido pelo(s) gestor(es) interessado(s).

ZooKeeper

O ZooKeeper [39] é um serviço centralizado para guardar informação de um sistema distribuído. É usado pelo kafka para centralizar a informação sobre o estado do nós do kafka *cluster*, sobre os tópicos, partições, e outros dados relevantes ao funcionamento do serviço kafka.

Em todos os nós onde são lançados contentores com o serviço kafka, é lançado outro contentor com o serviço ZooKeeper.

4.2.5 Regiões suportadas

Neste trabalho foram consideradas um total de sete regiões, representadas na figura 4.5: Europa, América do Norte, América do Sul, África, Médio Oriente, Ásia, e Oceânia. As regiões são determinadas pela sua coordenada (latitude, longitude), e um componente pertence a uma certa região consoante a sua proximidade ao ponto central dessa mesma região.

Em cada região existe a possibilidade de ser lançado um Balanceador de carga, um Servidor de registo, um Agente Kafka, e um Gestor local. Esses componentes de apoio estão lançados e removidos consoante a existência de contentores numa dada região.

Figura 4.5: Regiões definidas no sistema de gestão.

4.2.6 Dockerfiles e imagens docker

Para cada componente do sistema e micro-serviço suportado, foi desenvolvido um dockerfile, exemplos nas listagens 4.15 e 4.16, que define a imagem docker usada para lançar o serviço dentro de contentores. O DockerHub usmanager contém 41 repositórios, incluindo 11 imagens de componentes do sistema, 13 imagens da aplicação *Sock shop*, e

[39] Apache ZooKeeper: https://zookeeper.apache.org

17 imagens da aplicação *Hotel reservation*. Cada repositório no DockerHub está ligado ao respetivo repositório no GitHub, para ativar *builds* automaticamente sempre que haja uma alteração no código.

Listagem 4.15: Dockerfile do gestor local, usado para construir a sua imagem docker.

```
FROM maven:3.6.0-jdk-11-slim AS build
# Construir e instalar o JAR com o código da definição da base de dados
WORKDIR /usr/src/app/manager-database
COPY manager-database/src ./src
COPY manager-database/pom.xml ./pom.xml
RUN mvn install -DskipTests -U
# Construir e instalar o JAR com o código relativo aos serviços comuns aos gestores
WORKDIR /usr/src/app/manager-services
COPY manager-services/src ./src
COPY manager-services/pom.xml ./pom.xml
RUN mvn install -DskipTests -U
# Construir o JAR do gestor local
WORKDIR /usr/src/app
COPY manager-worker/src src
COPY manager-worker/pom.xml .
RUN mvn -DskipTests clean package
# Construir a imagem final
FROM adoptopenjdk/openjdk11:ubuntu-jre
COPY manager-worker/src/main/docker/scripts scripts
RUN sh scripts/dockerfile-docker-install.sh scripts/dockerfile-docker-api-install.sh
VOLUME /var/run/docker.sock
WORKDIR /app
COPY --from=build /usr/src/app/target/*.jar manager-worker.jar
EXPOSE 8081
ENTRYPOINT ["java","-jar","manager-worker.jar"]
```

Listagem 4.16: Dockerfile do micro-serviço *Frontend* da aplicação *Hotel reservation*.

```
FROM golang:1.14.6-alpine AS build
RUN apk add --no-cache git
# Gerar o ficheiro binário
ARG path=/go/src/github.com/usmanager/microservices/hotelReservation
WORKDIR $path
COPY cmd/frontend cmd/frontend
COPY data data
COPY dialer dialer
COPY registry registry
COPY services services
COPY tracing tracing
COPY wrk2_lua_scripts wrk2_lua_scripts
COPY docker docker
COPY config.json go.mod Gopkg.toml ./
WORKDIR $path/cmd/frontend/
RUN go build -o frontend && \
mkdir /app && \
```

```
18 mv frontend /app/frontend && \
19 mv $path/docker/frontend/docker-init.sh /docker-init.sh
20 # Construir a imagem final
21 FROM usmanager/registration-client AS registration-client
22 FROM alpine:3.12.0
23 RUN apk add --no-cache git && \
24 mkdir /app
25 WORKDIR /app
26 COPY --from=build /app/frontend frontend
27 COPY --from=registration-client /app/registration-client .
28 COPY --from=build docker-init.sh docker-init.sh
29 RUN ["chmod", "+x", "docker-init.sh"]
30 ENTRYPOINT ["./docker-init.sh"]
31 # registration-server, external-port, internal-port, hostname, registration-client-port
32 # Os valores realmente usados são depois definidos pelo gestor
33 CMD ["127.0.0.1:8761", "5000", "5000", "127.0.0.1", "1906"]
```

4.2.7 Operações resilientes

Certas operações são envolvidas num ciclo com até 5 tentativas na execução de ações cruciais para o sucesso da operação. A espera entre tentativas é implementada seguindo um *backoff* gradual, como na seguinte função:

$$T(i) = i + 1 \qquad i \in 0, \ldots, 4$$

Ou seja, após a primeira tentativa, é esperado 1 segundo, após a segunda tentativa são esperados 2 segundos, e assim sucessivamente. O ciclo com tentativas é implementado nos seguintes casos:

1. Obtenção das instâncias a executar na cloud.
2. Obtenção dos endereços IP elásticos alocados na Amazon.
3. Adição e configuração de um novo nó.
4. Lançamento de uma aplicação.
5. Lançamento de um contentor.
6. Envio de um pedido do gestor principal para um gestor local.

4.2.8 Assincronismo e paralelismo

O assincronismo e paralelismo foi implementado no sistema com recurso a diversas bibliotecas presentes na linguagem Java e Go, explicado na secções 4.2.8.1, 4.2.8.2 e 4.2.8.3. Entre as operações estão ciclos que executam periodicamente, operações assíncronas mas não paralelas, operações paralelas mas não assíncronas e operações assíncronas e paralelas.

4.2.8.1 Execuções periódicas

Em diversos componentes foram definidos vários ciclos que executam periodicamente.

Em componentes implementados na linguagem Java, é usado um objeto da classe *Timer* [40] do pacote java.util, como na listagem 4.17. O gestor principal inicia um total de três ciclos, relacionados com a monitorização dos nós, contentores e instâncias cloud. Um gestor local inicia quatro ciclos, dois para a sincronização dos dados sobre nós e contentores na base de dados em relação ao estado do seu *docker swarm*, e outros 2 ciclos para a monitorização de nós e de serviços.

Listagem 4.17: Exemplo da iniciação de um ciclo em Java, que executa periodicamente.

```
hostMonitoringTimer = new Timer("hosts-monitoring", true);
hostMonitoringTimer.schedule(new TimerTask() {
    @Override
    public void run() {
        try {
            monitorHostsTask();
        }
        catch (Exception e) {
            log.error("Failed to execute monitor hosts task: {}", e.getMessage());
            e.printStackTrace();
        }
    }
}, monitorPeriod, monitorPeriod);
```

Em componentes implementados na linguagem Go, é usada a função *NewTicker* do pacote *time* [41], como na listagem 4.18. O cliente de registo e descoberta de serviços define um ciclo que envia periodicamente ao monitor de pedidos, os registos de pedidos de descoberta de serviços.

Listagem 4.18: Exemplo da iniciação de um ciclo em Go, que executa periodicamente.

```
sendTicker := time.NewTicker(sendInterval)
stopChannel := make(chan bool)
go func(ticker *time.Ticker) {
    defer sendTicker.Stop()
    for {
        select {
        case <-ticker.C:
            sendRequestsData()
        case <-stopChannel:
            return
        }
    }
}(sendTicker)
```

[40] Classe timer da biblioteca java: https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html
[41] Pacote time da biblioteca go: https://golang.org/pkg/time

4.2.8.2 Operações assíncronas

Nos gestores as operações assíncronas foram implementadas usando a anotação @Async [42] da *framework* Spring boot, retornando *void*, ou, nos métodos considerados assíncronos, mas onde o resultado deve ser conhecido em qualquer altura da execução do sistema, um objeto da classe *CompletableFuture*, como na listagem 4.19. O código apresentado implementa um método para obter todas as instâncias virtuais a executar em todas as regiões AWS, assincronamente. Adicionalmente, é também implementado um mecanismo de tentativas, no caso dos pedidos ao serviço externo da AWS falhar. Neste caso, é tentado obter a lista de instâncias três vezes, após isso, o método falha lançando uma exceção.

Listagem 4.19: Exemplo de um método assíncrono, implementado na *framework* Spring Boot.

```
1  public List<Instance> getInstances() {
2     List<Instance> instances = new LinkedList<>();
3
4     List<AwsRegion> regions = AwsRegion.getAwsRegions();
5     CompletableFuture<?>[] requests = new CompletableFuture[regions.size()];
6    int count = 0;
7    for (AwsRegion awsRegion : regions) {
8         CompletableFuture<?> future = CompletableFuture
9       .supplyAsync(() -> this.getInstances(awsRegion))
10      .thenAccept(instances::addAll)
11      .thenApply(CompletableFuture::completedFuture)
12      .exceptionally(t -> retryGetInstances(t, 0, awsRegion, instances))
13      .thenCompose(Function.identity());
14    requests[count++] = future;
15    }
16
17   CompletableFuture.allOf(requests).join();
18
19    return instances;
20 }
21
22 private CompletableFuture<Void> retryGetInstances(Throwable first, int retry, AwsRegion
      ↪ awsRegion,
23                         List<Instance> instances) {
24    if (retry >= 3) {
25       return CompletableFuture.failedFuture(first);
26    }
27    return CompletableFuture
28       .supplyAsync(() -> this.getInstances(awsRegion))
29       .thenAccept(instances::addAll)
30       .thenApply(CompletableFuture::completedFuture)
31       .exceptionally(t -> {
```

[42] Async da *framework* Spring: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html

```
32              first.addSuppressed(t);
33              return retryGetInstances(first, retry + 1, awsRegion, instances);
34          })
35          .thenCompose(Function.identity());
36  }
```

Nos componentes implementados em Go, o assincronismo é suportado pela própria linguagem, através de *goroutines*. Basta colocar a palavra-chave *go* antes da definição da função, para que o código seja executado concorrentemente, como na seguinte listagem 4.20:

Listagem 4.20: Exemplo de uma execução assíncrona na linguagem Go.

```
1  go func() {
2      errc <- http.ListenAndServe(fmt.Sprintf(":%d", s.Port), router)
3  }()
```

4.2.8.3 Operações paralelas

Nos gestores, qualquer operação que envolva execuções a APIs externas são feitas em paralelo com o uso de um conjunto de classes que implementam as interfaces *Future* e *Executor* da biblioteca do Java, nomeadamente:

1. Os pedidos do lançamento de contentores com componentes do sistema em regiões distintas, como servidores de registo, balanceadores de carga, gestores locais, prometheus, ou agentes kafka, são processados em paralelo porque cada região está logicamente separada em termos de recursos.

2. Mandar parar vários contentores simultaneamente, com o exemplo na listagem 4.21.

3. Na configuração de um nó, no lançamento dos contentores de apoio que executam em todos os nós.

4. Obtenção da lista de instâncias virtuais na *cloud*, e dos endereços IP elásticos alocados na Amazon.

5. Na recolha de métricas em cada instância do prometheus de cada nó, para a monitorização de hosts e serviços.

6. Obtenção dos servidores registados de cada serviço nos balanceadores de carga a executar nas diferentes regiões.

7. Obtenção dos dados sobre os pedidos externos guardados no monitor de pedidos de cada nó.

8. No pedido de sincronização dos contentores e nós em cada gestor local, por parte do gestor principal.

9. No pedido do lançamento de vários contentores em diferentes gestores locais.

A classe *CompletableFuture* permite criar uma lista com intenções de execuções de pedidos, que depois podem ser executados em paralelo. Por exemplo, ao processar um pedido do Hub de gestão para lançar vários contentores em regiões distintas, é criada uma lista com as intenções de execução dos pedidos aos gestores locais, que são depois executados em paralelo. O tempo de processamento depende do tempo de resposta individual de cada gestor, bem como o número de *threads* atribuída à paralelização.

O método *parallelStream* do Java usa uma pool de threads comum a toda a aplicação. Portanto, é preciso encapsular a sua execução num objeto *ForkJoinPool*, definindo o número de *threads*. O gestor principal está configurado para usar 4 *threads* e tamanho da *queue* de 100. Nos gestores locais, a configuração está feita para serem usados 2 *threads* e com tamanho da *queue* também 100.

Listagem 4.21: Exemplo de uma execução em paralelo para parar vários contentores em simultâneo.

```
List<DockerContainer> containers = getContainers();
new ForkJoinPool(threads).execute(() ->
    containers.parallelStream().forEach(container -> {
        String id = container.getId();
        HostAddress hostAddress = container.getHostAddress();
        stopContainer(id, hostAddress);
    })
);
```

4.2.9 Autenticação e segurança

Todos os gestores estão protegidos por autenticação básica, sendo a autenticação efetuada através de um utilizador e uma palavra-passe. Os utilizadores autorizados estão guardados na base de dados, juntamente com a palavra-passe cifrada com *bcrypt* [43]. Como o sistema é apenas um protótipo, e a segurança não foi o foco desta dissertação, apenas está pré-definido um utilizador com estatuto administrador.

No Hub de gestão, a autenticação é feita através da uma página de *login*. O pedido de autenticação é enviado para o gestor principal, e em caso de sucesso, a informação sobre o utilizador autenticado é guardada no *brower*, numa *cookie* com tempo de expiração de 1 ano.

Está também configurado um *proxy* com autenticação básica com o objetivo de evitar que a informação do *docker* disponível através do docker API em cada nó esteja acessível ao publico. Assim, o API pode estar disponível apenas na própria máquina, e o *proxy*, após autenticação, redireciona o pedido para a porta do API do *docker*, dando acesso à informação do mesmo.

[43] bcrypt: https://www.openbsd.org/papers/bcrypt-paper.pdf

4.2.10 Ambiente do sistema de gestão

As instâncias virtuais na *cloud*, nós e contentores nos *docker swarms* geridos pelo sistema de gestão contém uma propriedade associada automaticamente pelo sistema, para serem distinguidas de entidades exteriores. No caso das instâncias virtuais na *cloud*, é associada uma *tag* usmanager=true às instâncias iniciadas e geridas pelos gestores. No caso dos nós e contentores nos *docker swarms* de cada gestor, é associada uma *label* usmanager=true. Depois nos ciclos de sincronização, descritos anteriormente na secção 4.2.3, as entidades que não contém a propriedade, são ignoradas. O que permite que existam entidades que não pertençam ao sistema de gestão, mas que executem no mesmo ambiente dos gestores.

Capítulo 5

Demonstração, validação e avaliação experimental

Na demonstração 5.1 são apresentadas as secções que compõem o Hub de gestão, apresentando figuras da interface gráfica. A validação e avaliação experimental foi feita com recurso a testes unitários 5.2, que garantem a continuação dos funcionamento do código em trabalhos futuros, e usando um conjunto de casos de estudo descritos na secção 5.3, foram obtidos os tempos de execução de várias operações do sistema 5.4.1, e feitos testes funcionais 5.4.2 e testes de carga 5.4.3 com base em cenários desenhados para testar o desempenho e o comportamento do sistema.

5.1 Demonstração do Hub de gestão

Como descrito anteriormente na secção 4.2.2.2, o hub de gestão é uma interface gráfica acessível através do *browser*, para permitir visualizar e interagir com o sistema. Tem a seguinte composição:

- **Página principal**. Na figura 5.1 é possível visualizar uma configuração hipotética do sistema:
  a) Dois nós a executar na América do Norte (Estados unidos - oeste e este);
  b) Quatro nós a executar na Europa (Portugal, França e dois na Alemanha). Mostrando a informação relativa aos nós localizados em Frankfurt, na Alemanha;
  c) Dois nós a executar na Ásia (Índia e Japão);
  d) Um nó a executar na Oceania (Austrália).

- **Aplicações**. Na secção das aplicações, figura 5.2, é apresentada uma lista das aplicações registadas no sistema.

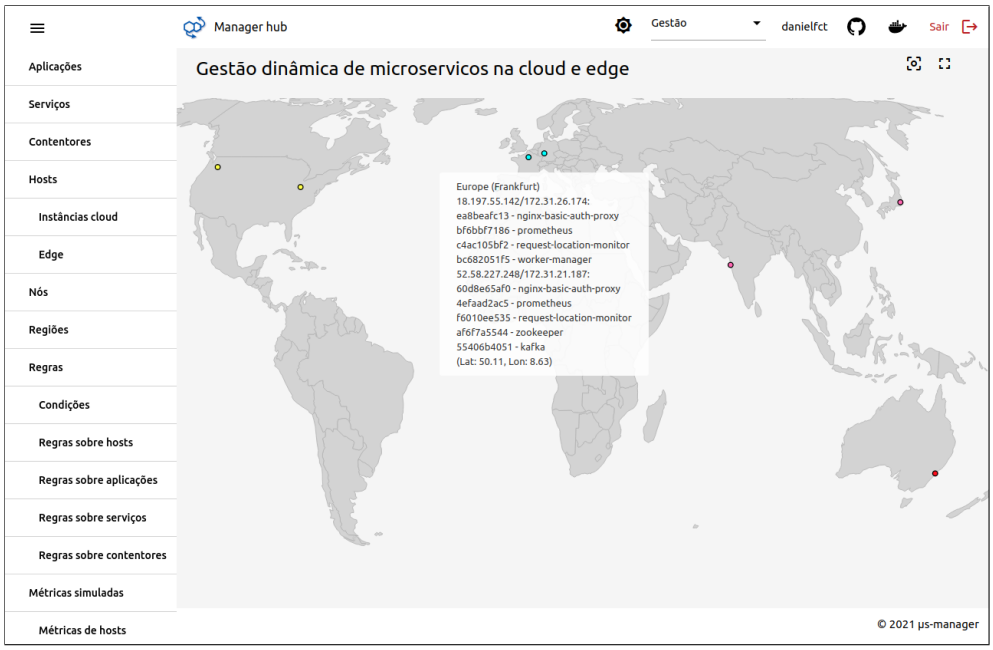


Figura 5.1: Mapa interativo presente na página principal do Hub de gestão.

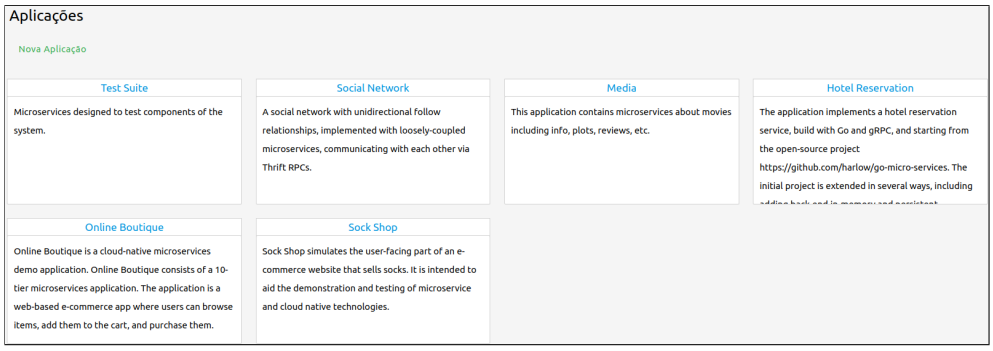


Figura 5.2: Lista de aplicações registadas no sistema.

Ao clicar na aplicação *Sock shop*, é mostrada uma página com o conteúdo da figura 5.3.

- **Serviços**. Na secção dos serviços, é apresentada a lista de serviços de todas as aplicações no sistema, visível na figura 5.4.

  Clicando no serviço *frontend* da aplicação *Sock shop*, é apresentada a página da figura 5.5.

- **Contentores**. Em relação aos contentores, é apresentada a lista de todos os contentores conhecidos pelo gestor principal, visível na figura 5.6.

  Existem duas possibilidades para lançar um contentor, figura 5.7: por localização e numa máquina especifica.

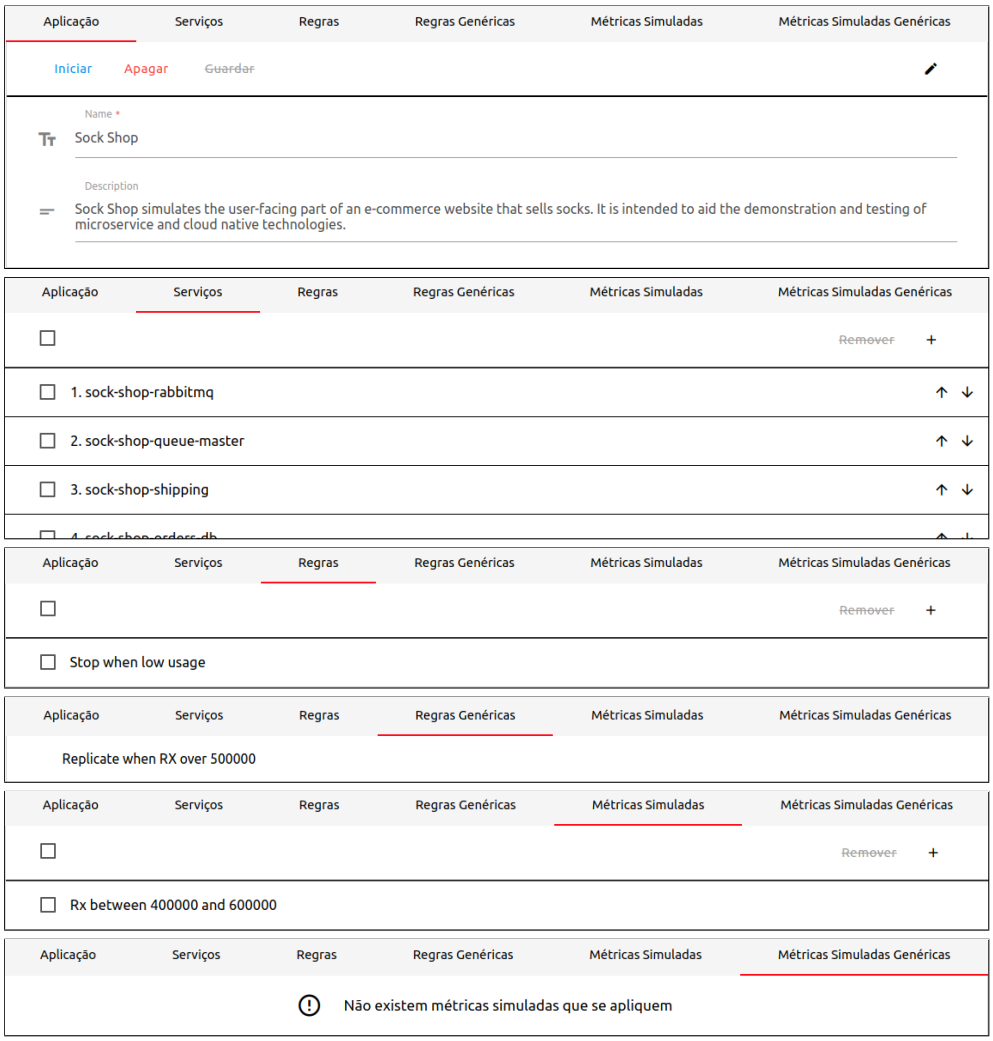


Figura 5.3: Detalhes sobre a aplicação *Sock Shop*.

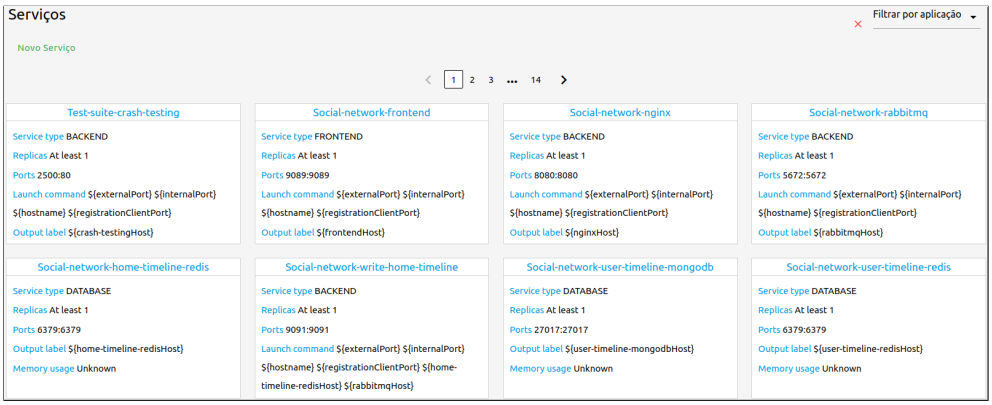


Figura 5.4: Lista de serviços registados no sistema.

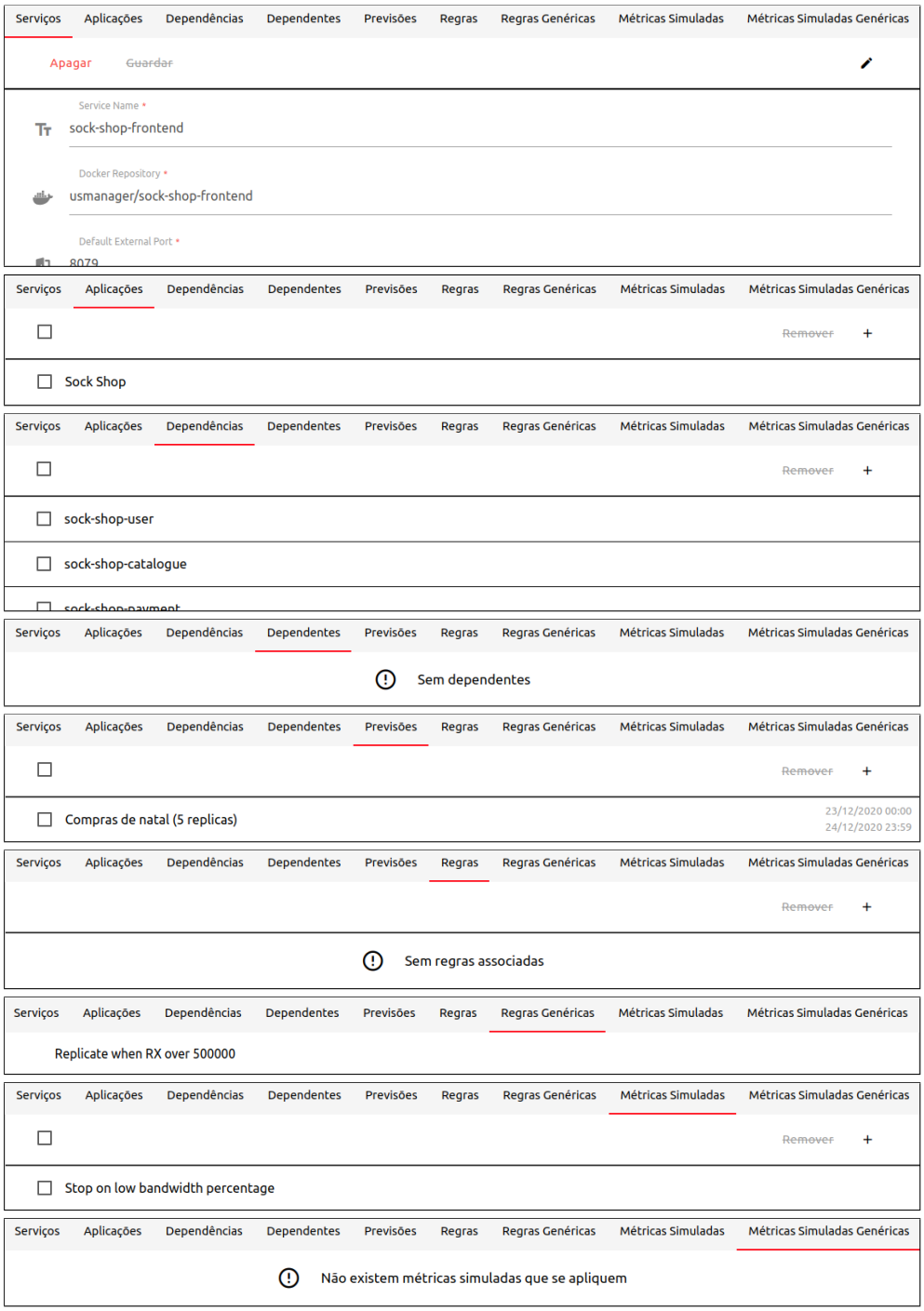


Figura 5.5: Detalhes do serviço *Frontend* da aplicação *Sock Shop*.

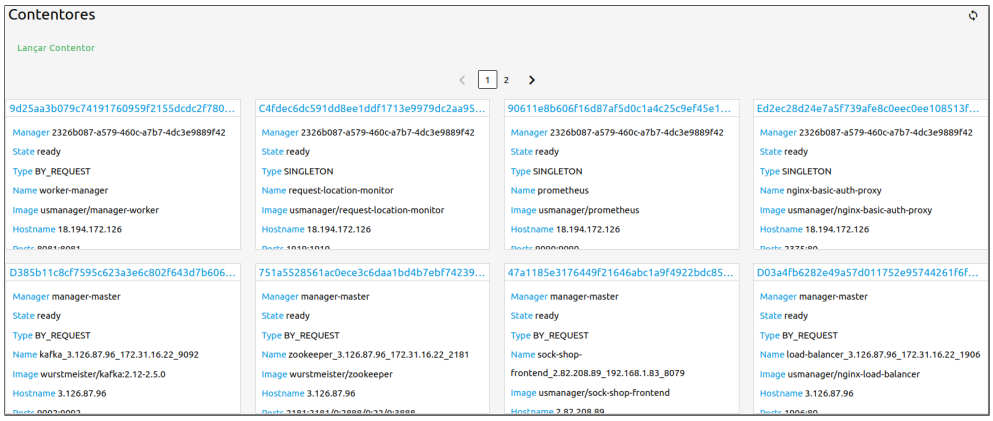
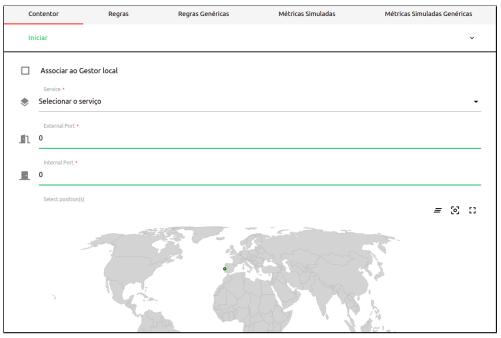
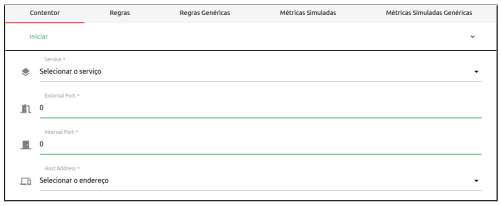


Figura 5.6: Lista de contentores reconhecidos pelo gestor principal.



(a) Usando coordenadas.



(b) Usando um endereço.

Figura 5.7: Formulários para iniciar um contentor.

Ao clicar num contentor do serviço *frontend* da aplicação *Sock shop*, é mostrada a figura 5.8.

- **Hosts**. A secção dos *hosts* está dividida em instâncias virtuais na *cloud* e máquinas registadas na periferia da rede (*edge*), visualizada na figura 5.11.

  A figura 5.10 mostra a página onde é possível iniciar uma nova instância virtual na *cloud*, através da seleção das coordenadas, num mapa interativo que inclui a localização de todas as zonas suportadas. Registar uma máquina na periferia da rede é simplesmente preencher e submeter toda a informação necessária, num formulário semelhante à segunda imagem da figura 5.11.

  Ao clicar num dos *hosts*, figura 5.11, é possível ver os detalhes do *host*, gerir as regras associadas, ver a lista das regras genéricas aplicadas aos *hosts*, associar métricas simuladas, e ver a lista das métricas simuladas genéricas. Existem ainda duas secções, uma para executar comandos SSH e outra para carregar ficheiros no *host*, através de SFTP.

- **Nós**. Na página dos nós, figura 5.12, é apresentada a lista de todos os nós conhecidos

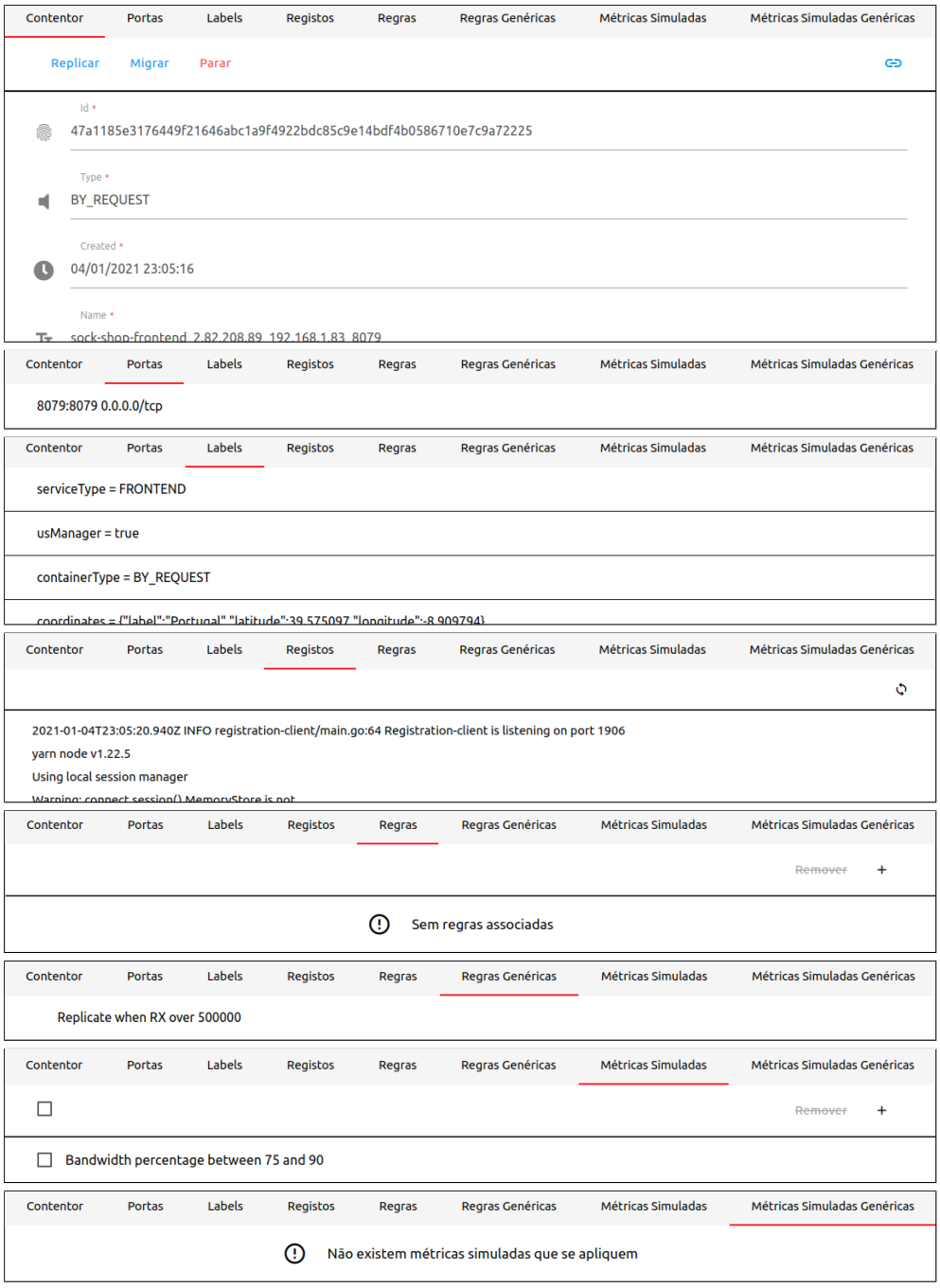


Figura 5.8: Detalhes do contentor *Frontend* da aplicação *Sock Shop*, a executar num nó.

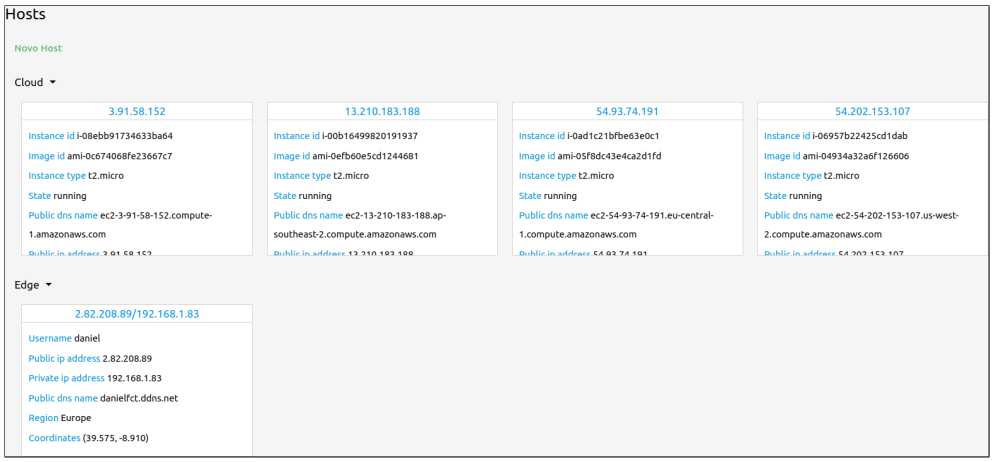


Figura 5.9: Página com a lista dos *hosts* geridos pelo sistema.

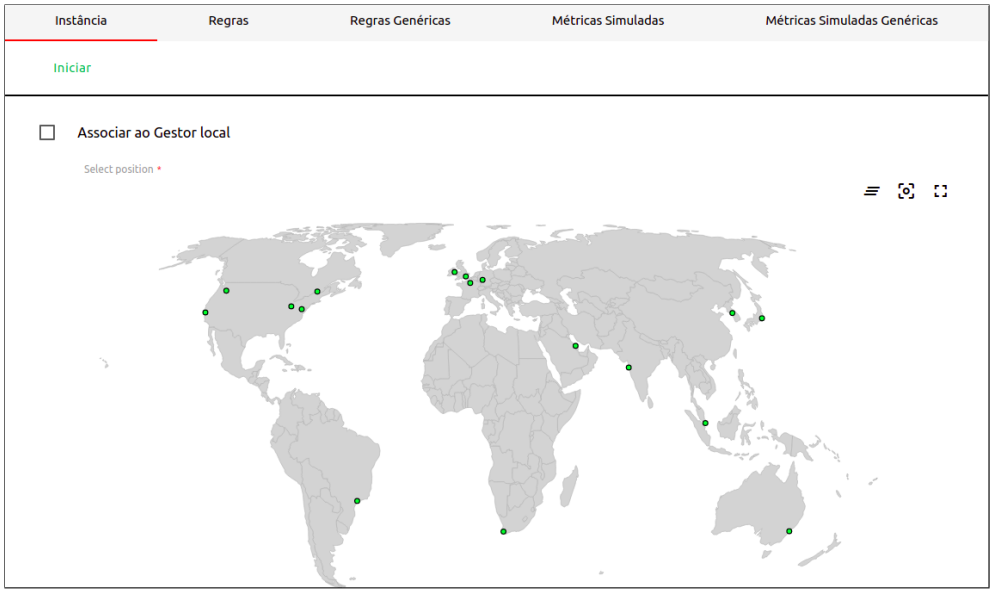


Figura 5.10: Mapa interativo para iniciar novas instâncias virtuais na *cloud*.

pelo gestor principal.

Ao clicar num nó da máquina 35.158.96.24 é apresentada uma página visível na figura 5.13, onde são mostrados os seus detalhes, juntamente com as *labels* associadas ao nó.

- **Regiões**. Nesta secção são apresentadas as regiões suportadas pelo sistema, visível na figura 5.14.

- **Regras**. A secção das regras, figura 5.15, está dividida em cinco partes: condições, regras aplicadas a *hosts*, regras aplicadas a aplicações, regras aplicadas a serviços e regras aplicadas a contentores.

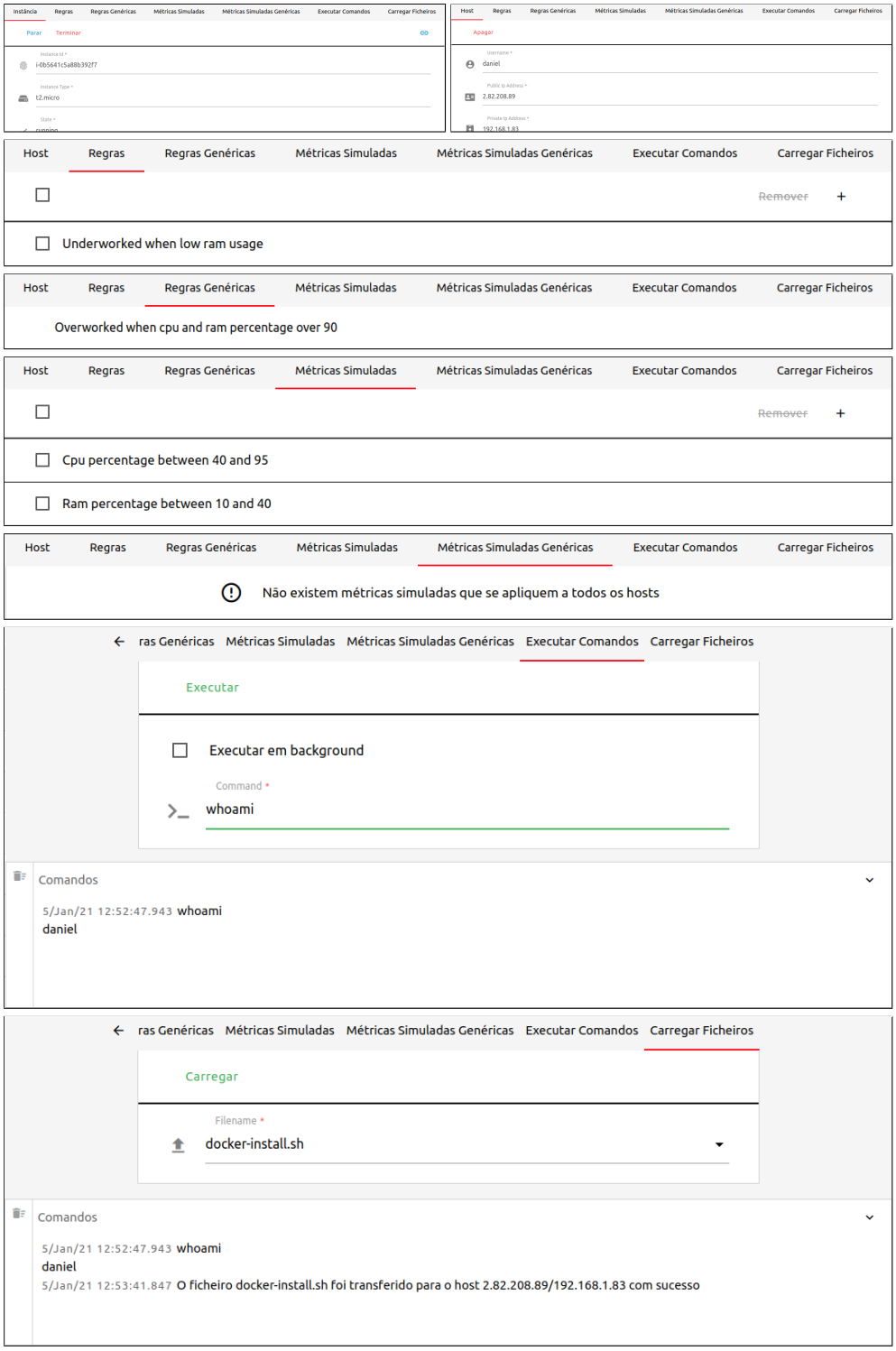


Figura 5.11: Detalhes de um *host* na *cloud* ou na *edge*.

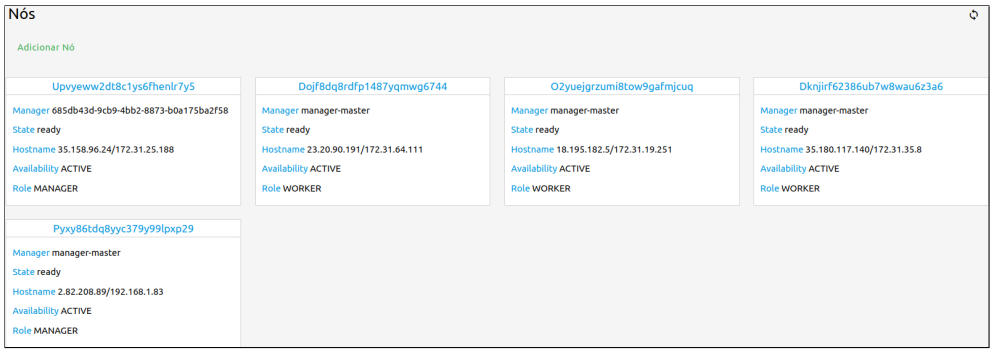


Figura 5.12: Lista de nós registados no gestor principal.

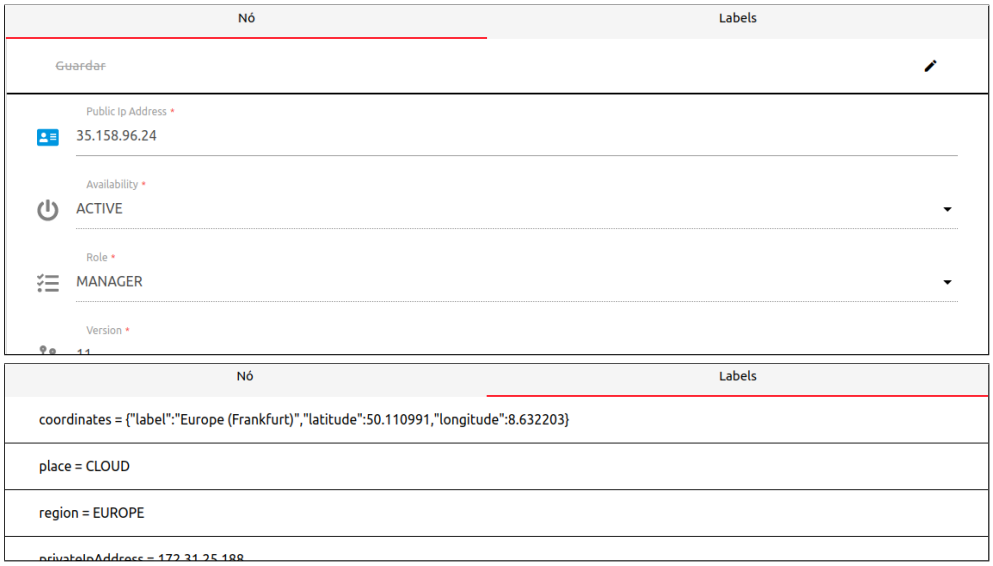


Figura 5.13: Detalhes de um nó do sistema.

Ao clicar na condição *"Tx bytes per second over 100000"*, é possível ver, editar, ou apagar a informação relativa à condição, como é visível na figura 5.16.

Ao seguir a regra *"Cpu and ram over 90"*, é apresentada uma página para editar os detalhes da regra, bem como associar entidades à mesma. No caso das regras aplicadas a *hosts*, como na figura 5.17, é possível associar condições, instâncias *cloud*, e máquinas *edge*. Semelhantemente, nas regras aplicadas a aplicações, serviços, ou contentores, é possível associar a respetiva entidade a que são aplicadas. Nas regras aplicadas a *hosts* e serviços, é possível definir a regra como sendo genérica, ou seja, é aplicada a todos os *hosts* ou serviços registados no sistema que ficam associados a essa regra.

- **Métricas simuladas**. Parecido às regras, a página das métricas simuladas, figura 5.18, inclui uma secção para ver as métricas simuladas aplicadas a *hosts*, aplicações, serviços e contentores.

  Após clicar na métrica simulada *"High bandwidth usage"*, a página muda para ser

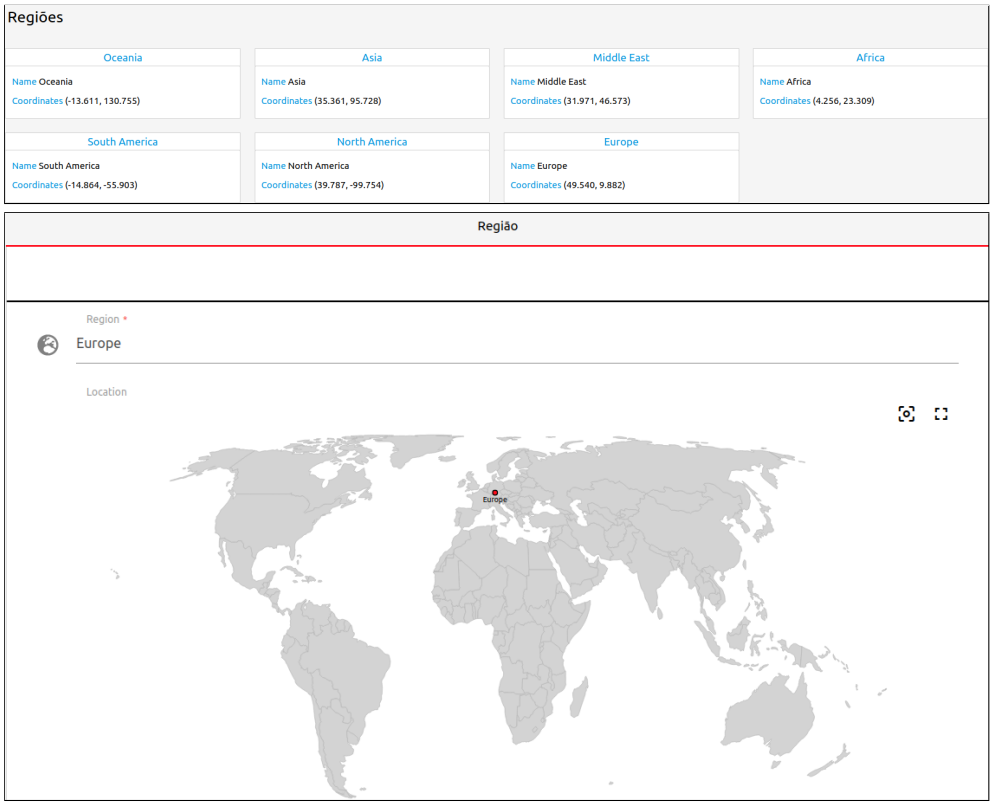


Figura 5.14: Regiões suportadas pelo sistema e detalhes da região Europa.

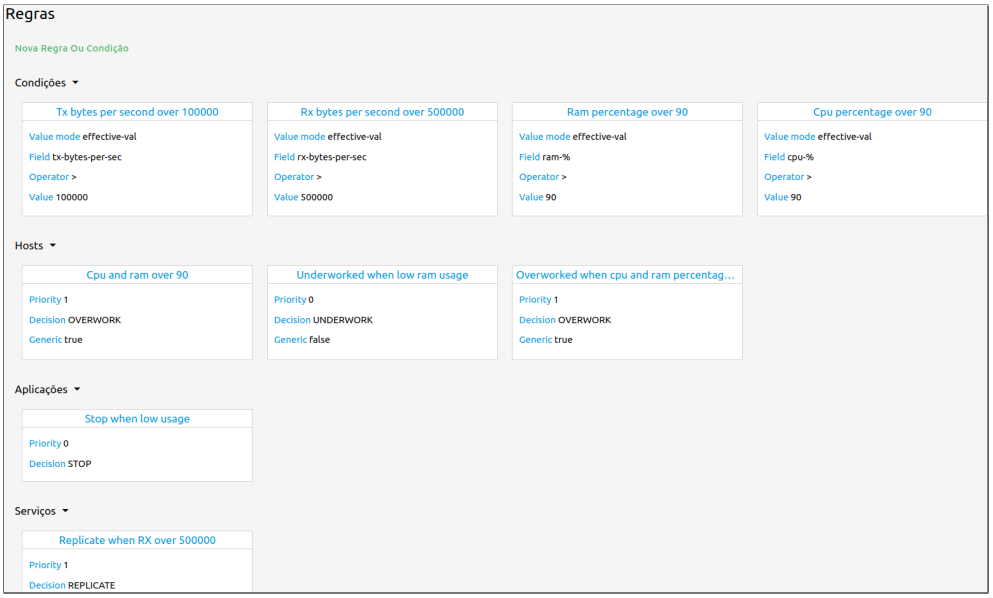


Figura 5.15: Lista de condições e regras registadas no sistema.

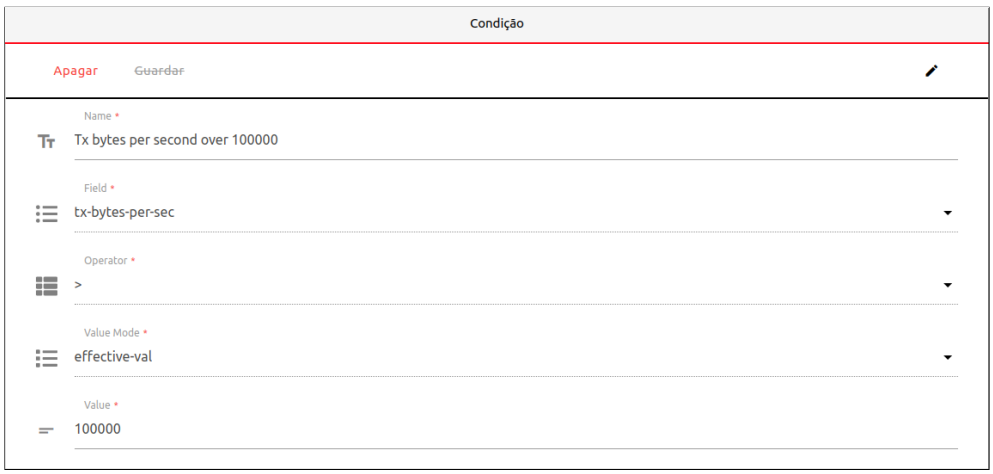


Figura 5.16: Detalhes de uma condição.

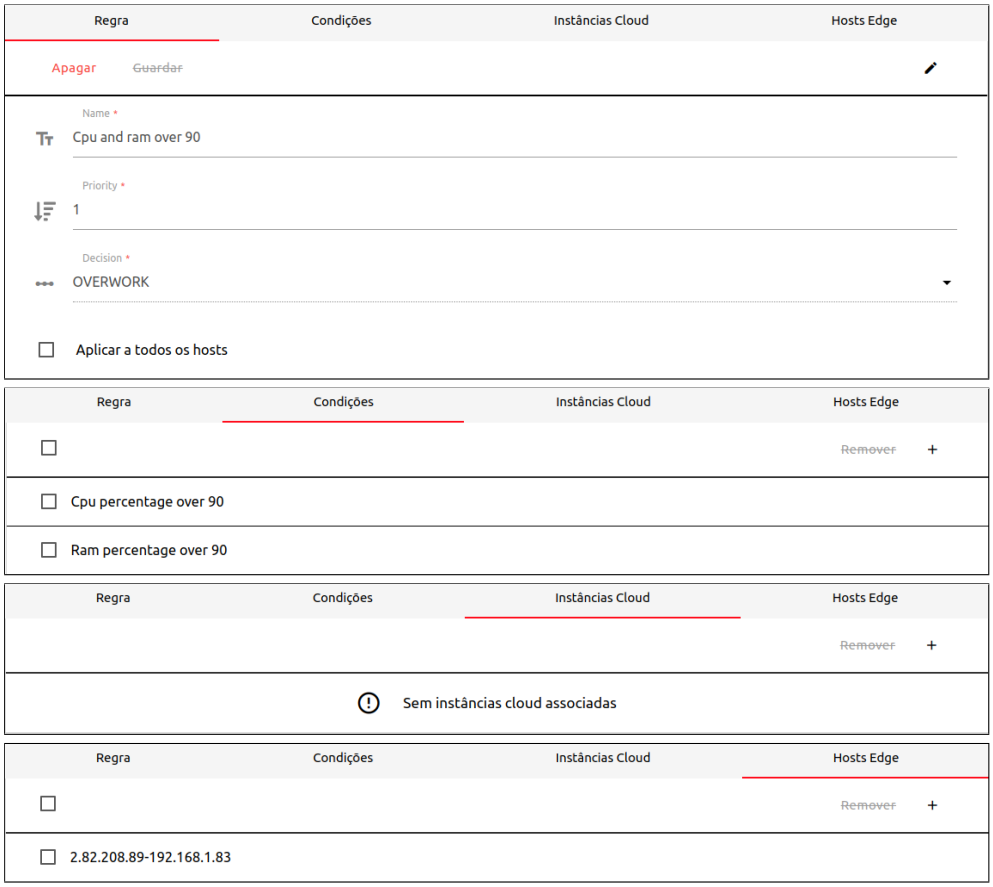


Figura 5.17: Detalhes de uma regra aplicada a *hosts*.

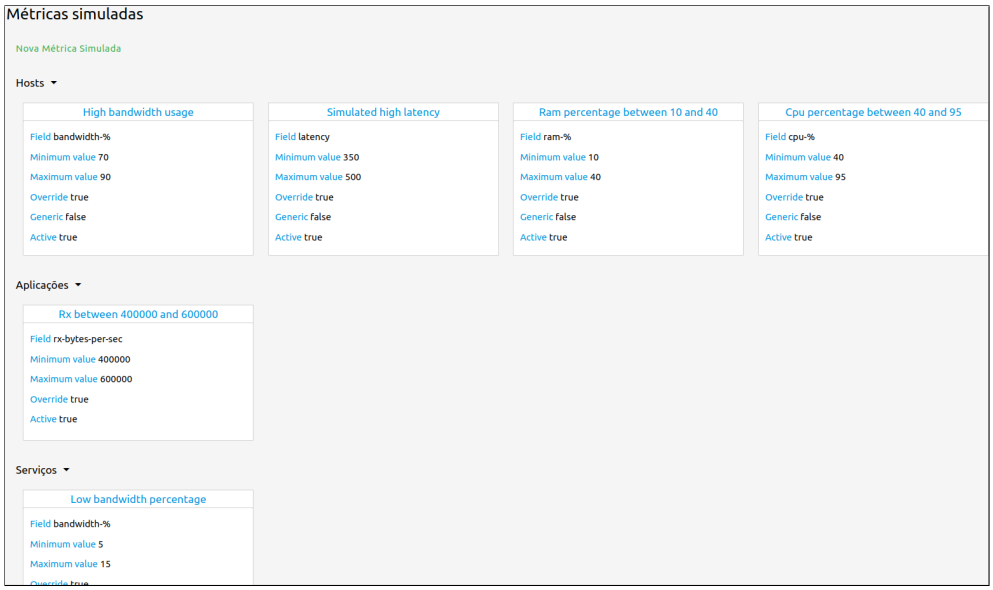


Figura 5.18: Lista de métricas simuladas registadas no sistema.

apresentada a informação relativa à métrica, incluindo secções para a associar as entidades. No caso de métricas simuladas aplicadas a *hosts*, figura 5.19, é possível associar *cloud* e *edge hosts* aos quais é aplicada a métrica. Semelhantemente, no caso de métricas simuladas aplicadas a aplicações, serviços, e contentores, é possível associar as respetivas entidades. Tal e qual como nas regras, nas métricas simuladas de *hosts* e serviços, também é possível definir a regra como sendo genérica.

- **Balanceamento de carga**. A parte do balanceamento de carga inclui uma lista com os balanceadores de carga a executar em cada região. Clicando numa balanceador, por exemplo o que executa na Europa, é possível aceder a mais detalhes sobre o mesmo, que pode ser visualizado na figura 5.20.

- **Servidores de registo**. Sobre os servidores de registo, é apresentada uma lista com todos os servidores a executar em cada região. Ao aceder a um dos servidores de registo, é apresentada uma página com os detalhes do servidor. Na figura 5.21 está o exemplo do servidor a executar na Europa.

- **Gestores locais**. A secção dos gestores locais lista todos os gestores locais lançados em cada região, e após clicar num dos gestores, como na figura 5.22, é apresentada a informação do respetivo gestor, incluindo as listas dos nós e contentores geridos por cada gestor.

- **Agentes Kafka**. Nesta secção estão os agentes *kafka*, com uma página para listar todos os agentes a executar nas diferentes regiões, e, após clicar num dos agentes, outra página para apresentar os detalhes, como é exemplificado na figura 5.23, com o agente da região Europa.

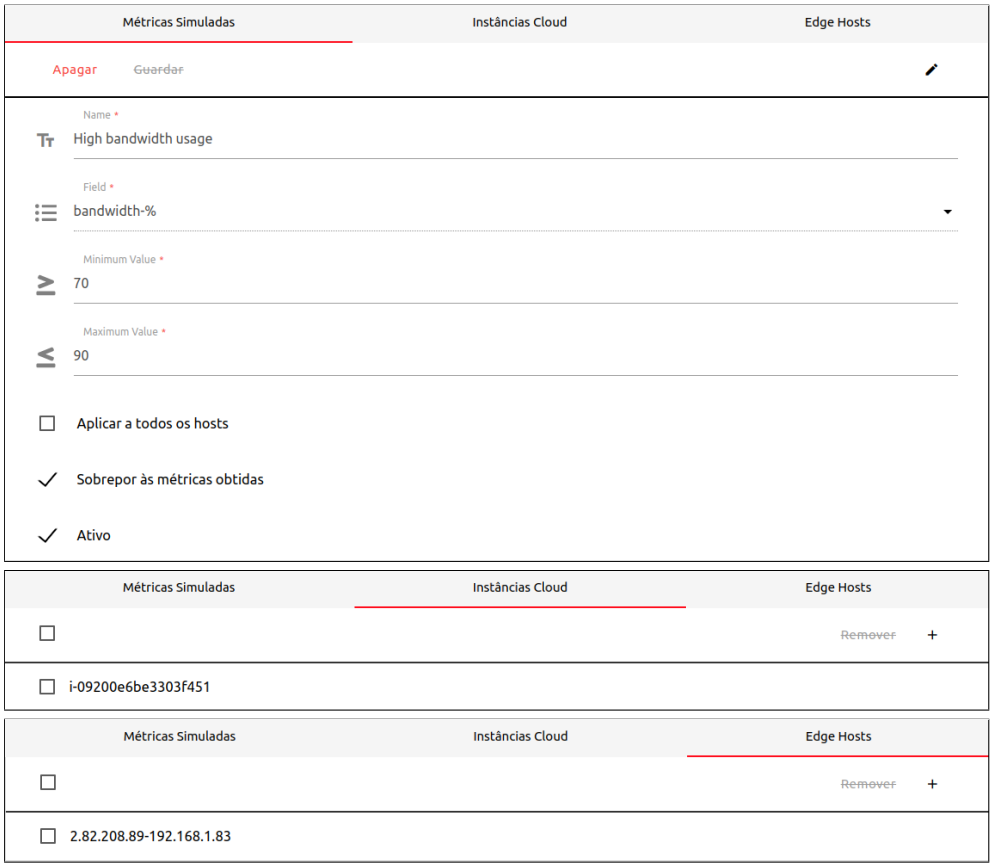


Figura 5.19: Detalhes de uma métrica simulada aplicada a *hosts*.

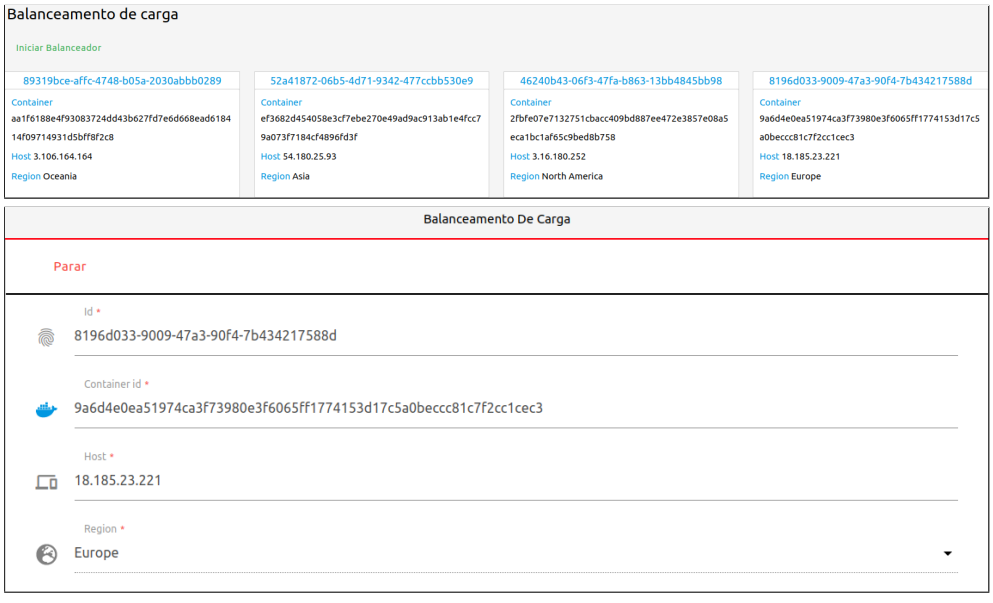


Figura 5.20: Secção dos balanceadores de carga registados no sistema.

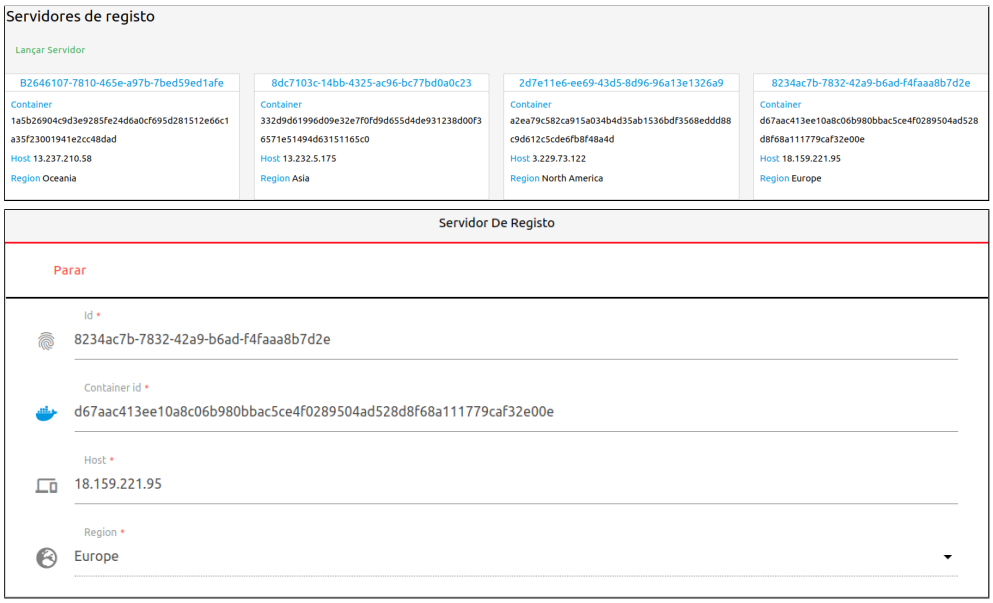


Figura 5.21: Secção dos servidores de registo a executar.

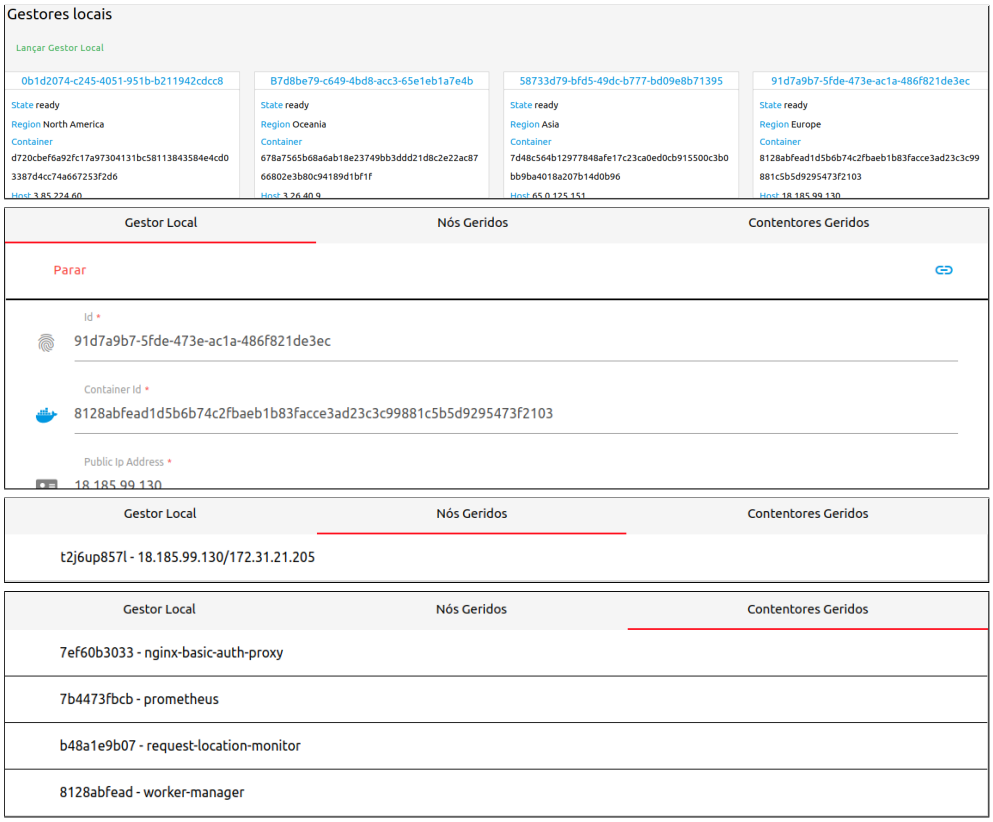


Figura 5.22: Secção dos gestores locais de cada região, e detalhes do gestor a executar na Europa.

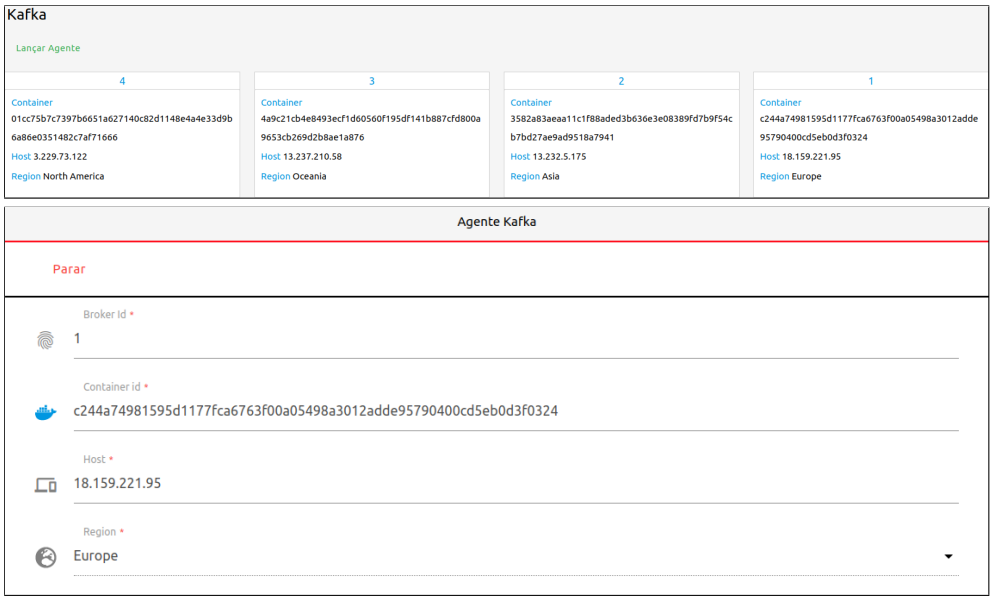


Figura 5.23: Secção dos agentes Kafka a executar em cada região, e detalhes do agente na Europa.

- **Secure shell**. Nesta secção da aplicação é possível executar comandos SSH e, através do protocolo SFTP, carregar ficheiros nos nós do sistema. Um exemplo pode ser visto na figura 5.24.
- **Registo**. A página dos registos, figura 5.25, inclui os *logs* do gestor principal. No exemplo é possível ver as execuções da sincronização de contentores e nós do *docker swarm* e das instâncias virtuais na AWS com a informação na base de dados. E também a monitorização de *hosts*, com o registo das métricas obtidas num dos *hosts*.

5.2 Testes unitários

Foram desenvolvidos testes unitários para testar partes fundamentais dos gestores que permitem garantir que alterações futuras ao código e/ou às funcionalidades dos gestores, não afetem as funcionalidades anteriormente implementadas e testadas. Na listagem 5.1 é incluído um exemplo de um teste unitário feito ao mapeamento de um objeto DTO para uma entidade JPA da base de dados, funcionalidade que é usada na comunicação 3.6.0.1 entre o gestor principal e os gestores locais no sistema.

Listagem 5.1: Teste Unitário para testar o mapeamento de DTOs em entidades JPA.

```
1 @Test
2 public void testMapDtoToEntity() {
3     ServiceDTO serviceDto = new ServiceDTO("service");
4     ServiceDTO dependencyDto = new ServiceDTO("dependency");
5     ServiceDependencyKey key = new ServiceDependencyKey(
6         serviceDto.getName(), dependencyDto.getName());
7     ServiceDependencyDTO serviceDependencyDto =
```

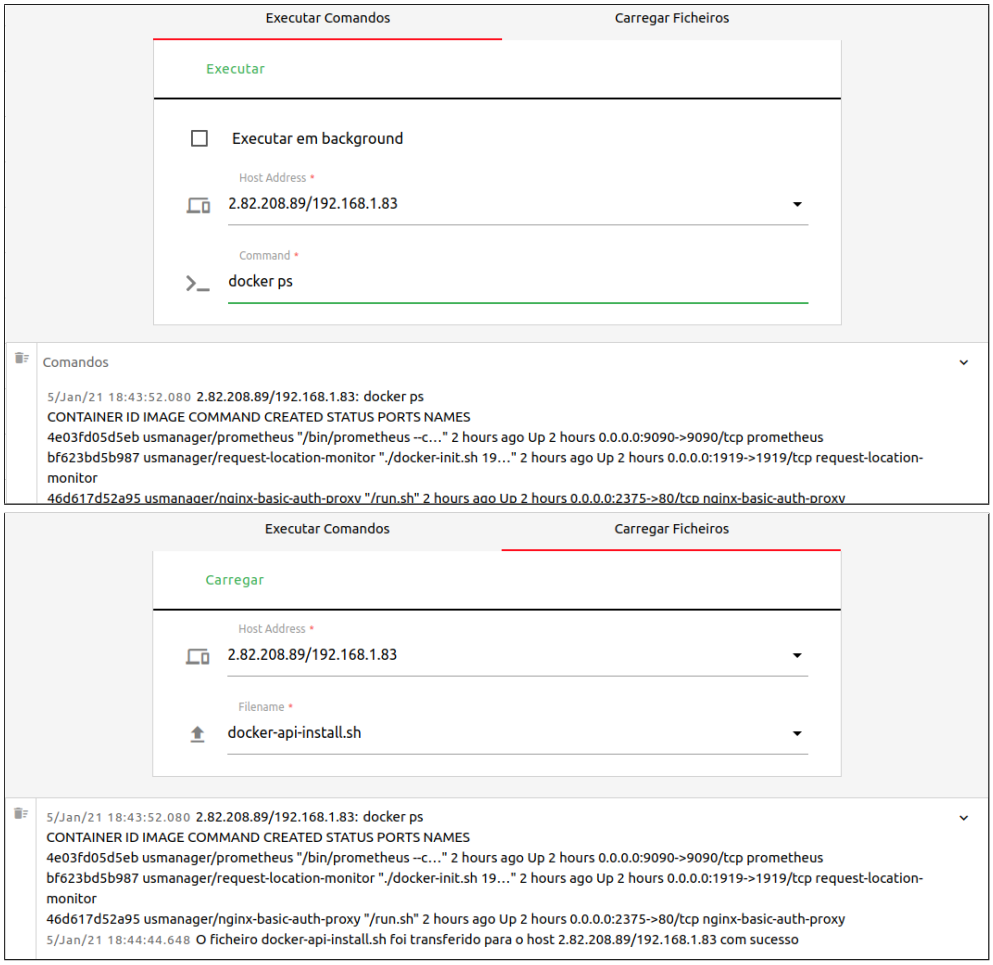


Figura 5.24: Página para executar comandos SSH e carregar ficheiros através do protocolo SFTP.

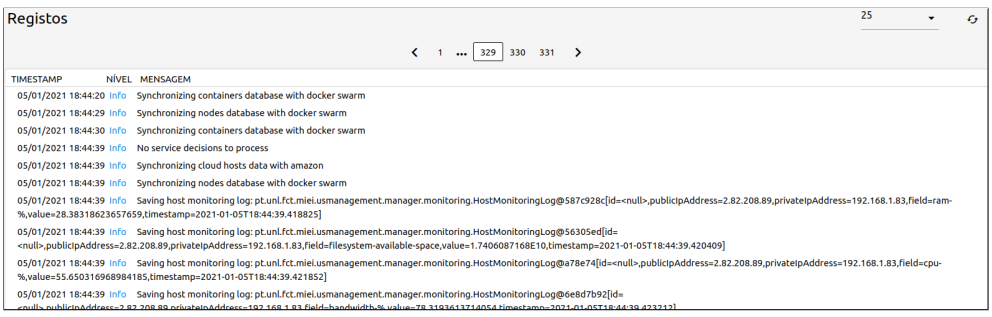


Figura 5.25: Página com os registos do gestor principal.

```
8           new ServiceDependencyDTO(key, serviceDto, dependencyDto);
9       serviceDto.setDependencies(Set.of(serviceDependencyDto));
10      CycleAvoidingMappingContext context =
11          new CycleAvoidingMappingContext();
12      Service service = ServiceMapper.MAPPER.toService(serviceDto,
13          context);
14      assertThat(service).isNotNull();
15      assertThat(service.getName()).isEqualTo("service");
16      Set<ServiceDependency> serviceDependencies =
17          service.getDependencies();
18      assertThat(serviceDependencies).hasSize(1);
19      assertThat(serviceDependencies).extracting("service")
20          .containsExactly(service);
21  }
```

5.3 Casos de estudo

Para a validação do sistema foram consideradas duas aplicações, a *Sock shop* e a *Hotel Reservation*. A aplicação *Sock shop* [1] foi desenvolvida pela *Weaveworks* para simular um *website e-commerce* que vende meias. Esta foi a aplicação que foi adaptada e usada como caso de estudo nos testes do trabalho anterior. Continua a estar presente no sistema, e a ser novamente usada como caso de estudo. A aplicação *Hotel Reservation* foi ajustada para usar a funcionalidade de descoberta de serviços com recurso ao cliente e ao servidor de registo e descoberta de serviços. O que implicou alterar o *dockerfile* que gera a imagem *docker* de cada serviço, para incluir na imagem o componente de descoberta de serviços. Também foi necessário modificar o código dos micro-serviços com dependências, para que o *endpoint* de comunicação a serviços externos seja dinâmico.

De notar que, com esta solução de descoberta de serviços, estes não podem ter estado de sessão relativo a outros serviços. Ou seja, a comunicação deve ser *stateless*, visto que o *endpoint* com que comunicam pode não ser sempre o mesmo.

Os dados de outras quatro aplicações foram adicionadas à base de dados do sistema: *Social Network*, *Media*, *Online Boutique* e *Testing suite*. Mas os seus serviços não foram modificados para usarem a funcionalidade de descoberta de serviços, e portanto não foram usadas durante os testes.

As aplicações *Hotel Reservation*, *Social Network* e *Media* pertencem ao projeto Death Star Bench [2] [10] que está a ser implementado por uma equipa na *Cornell University*, com o objetivo de disponibilizar, em open-source, aplicações compostas por micro-serviços para que possam ser usadas como teste. À data da escrita deste documento, encontravam-se 3 serviços totalmente implementados, sendo que outros 3 (*E-commerce site*, *Banking System*, *Drone coordination system*) ainda estavam em desenvolvimento, e portanto não incluídos no sistema.

[1] Repositório da aplicação Sock shop: https://microservices-demo.github.io
[2] Projeto Death Star Bench: https://github.com/delimitrou/DeathStarBench

A aplicação *Online Boutique* [3] foi desenvolvida por uma equipa pertencente à Google, com o objetivo de ser usada como demo para demonstrar tecnologias associadas à *cloud*.

A aplicação *Testing suite* foi desenvolvida de raiz, a qual inclui serviços desenvolvidos especificamente para testarem funcionalidades do sistema.

5.3.0.1 Sock Shop

A aplicação *Sock Shop* implementa um sistema para disponibilizar um *website e-commerce* de venda de meias. É composta por 9 micro-serviços implementados em Java e Go, com a arquitetura da aplicação disponível na figura 5.26.

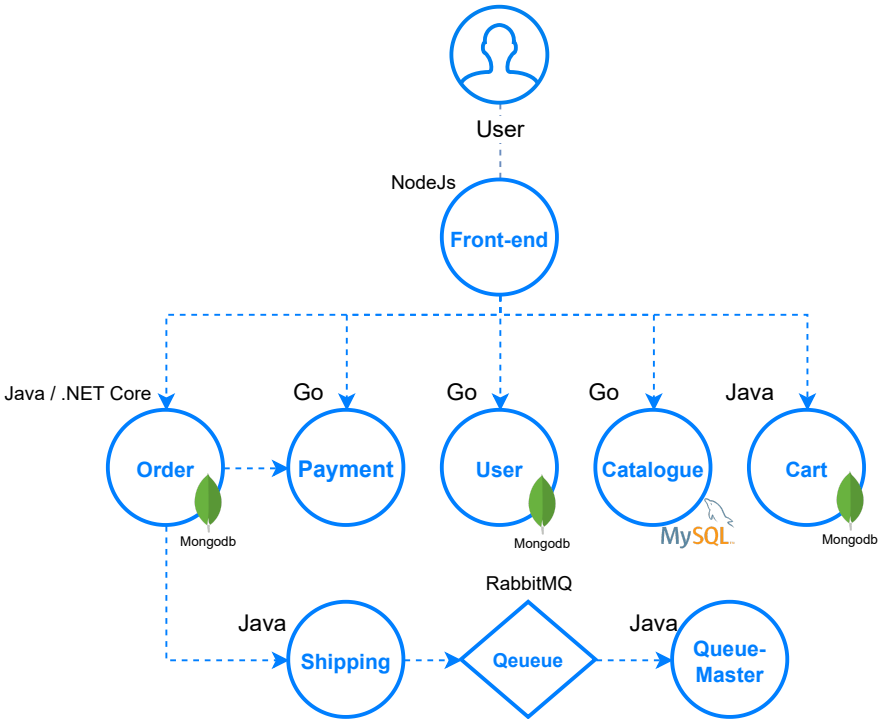


Figura 5.26: Arquitetura da aplicação *Sock Shop*.

5.3.0.2 Hotel Reservation

A aplicação *Hotel Reservation* [4], com a sua arquitetura demonstrada na figura 5.27, é composta por 8 micro-serviços que implementam um serviço de reserva de quartos de hotéis. A linguagem usada para implementar os serviços foi o Go [5], sendo que para a comunicação entre serviços, foi usada a *framework* gRPC [6]. A principal funcionalidade da aplicação é a reserva de quartos de hotéis, mas também inclui serviços para recomendações, avaliações, e procura de hotéis.

[3] Repositório da aplicação Online Boutique: https://github.com/GoogleCloudPlatform/microservices-demo

[4] Hotel Reservation: https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation

[5] Linguagem Go: https://golang.org

[6] gRPC: https://grpc.io

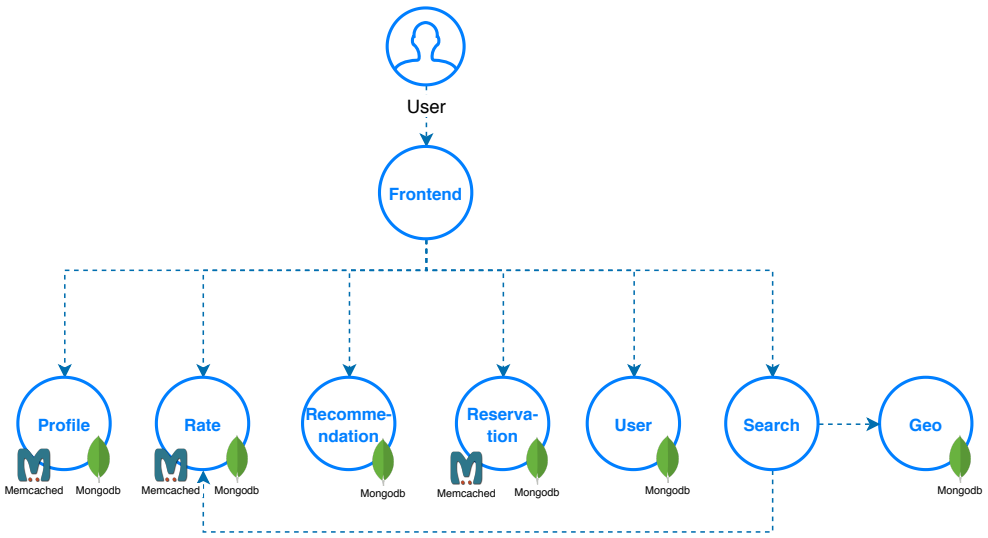


Figura 5.27: Arquitetura da aplicação *Hotel Reservation*.

5.3.0.3 Social Network

A aplicação *Social Network*, é composta por 13 micro-serviços, todos escritos em C++. Alguns micro-serviços têm dependências a outros serviços e/ou a base de dados e/ou ao *memcached* [7]. A sua arquitetura pode ser vista na imagem 5.28. As principais funcionalidades são:

1. Registar e autenticar usando as credenciais do utilizador;
2. Criar informação em formato texto, imagem, vídeo, URL ou *tags* de utilizadores;
3. Ver informação sobre *posts*;
4. Procurar por um utilizador ou informação na base de dado;
5. Ver a linha do tempo de um utilizador;
6. Receber recomendações sobre quais utilizadores seguir;
7. Seguir e deixar de seguir utilizadores.

5.3.0.4 Media

A aplicação *Media* [8] é composta por 13 micro-serviços programados na linguagem C++ [9], e implementa um sistema de criticas a filmes e series. A sua arquitetura pode ser vista na figura 5.29, onde pode ser observado que alguns serviços fazem uso de persistência

[7] Memcached: https://memcached.org
[8] Repositório da aplicação Media: https://github.com/delimitrou/DeathStarBench/tree/master/mediaMicroservices
[9] Linguagem C++: https://www.cplusplus.com

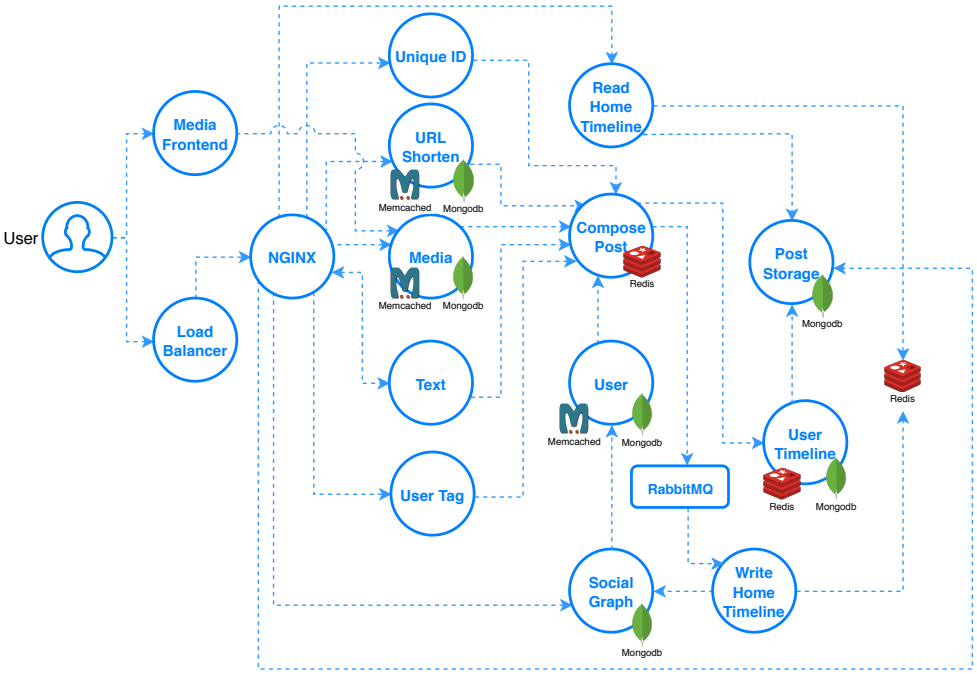


Figura 5.28: Arquitetura da aplicação *Social network*.

de dados através do Mongodb [10] ou Redis [11], bem como a utilização do memcached para aumentar a velocidade de acesso à base de dados de serviços cruciais.

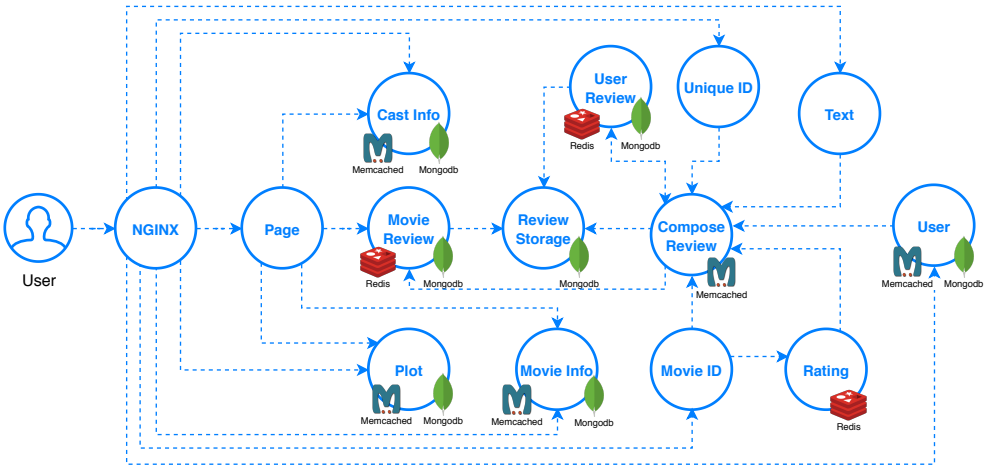


Figura 5.29: Arquitetura da aplicação *Media*.

[10] MongoDB: https://www.mongodb.com
[11] Redis: https://redis.io

5.3.0.5 Online Boutique

A aplicação *Online-boutique*, visualizada na figura 5.30, contém 11 micro-serviços escritos em diferentes linguagens. É o conjunto de micro-serviços que a Google usa para demonstrar tecnologias associadas à *cloud*, como Kubernetes/GKE [12], Istio [13], Stackdriver [14], gRPC [15] e OpenCensus [16].

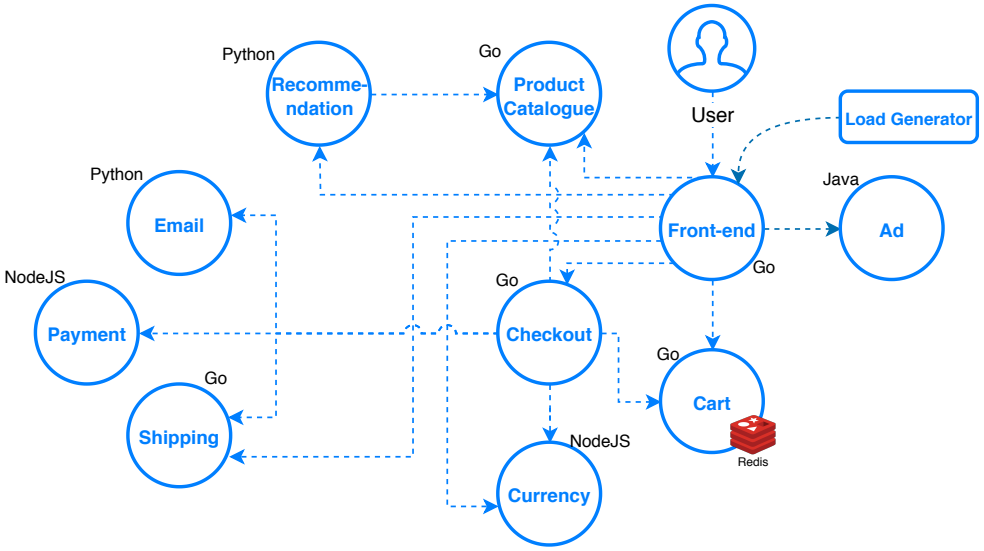


Figura 5.30: Arquitetura da aplicação *Online Boutique*.

5.3.0.6 Testing suite

A aplicação *Testing suite* contém micro-serviços desenhados especificamente para testar funcionalidades do sistema. Atualmente apenas existe um micro-serviço, chamado *crash-testing*, com o objetivo de abortar propositadamente e testar a funcionalidade de recuperação de contentores, que acabou por não ser implementada na totalidade, e portanto não foi referida anteriormente no documento.

5.4 Avaliação experimental

A avaliação experimental tem como objetivo, através da execução das funcionalidades do sistema, comprovar o correto funcionamento do sistema, medindo o seu desempenho e observando o seu comportamento quando confrontado com carga variável por parte de utilizadores virtuais.

[12] Kubernetes/GKE: https://cloud.google.com/kubernetes-engine
[13] Istio: https://istio.io
[14] Stackdriver: https://cloud.google.com/products/operations
[15] gRPC: https://grpc.io
[16] OpenCensus: https://opencensus.io

5.4.1 Tempos de execução

Para se obter uma ideia do desempenho do sistema, foram obtidos vários tempos de execução das operações fundamentais do sistema, desde a iniciação, configuração e terminação de instâncias virtuais na *cloud*, lançamento de componentes de apoios, lançamento de contentores de serviços, à obtenção de recursos externos ao sistema, como por exemplo à AWS. Os tempos foram obtidos usando um *script* visível na listagem 5.2, que é executado pela ferramenta de testes k6 [17], produzindo um conjunto de métricas, onde está incluído o tempo de execução do pedido. Através de argumentos, é possível indicar o número de execuções da operação, o endereço da máquina onde está a executar o gestor principal, e o tipo de método, o URL e o corpo do pedido que é usado no pedido REST. O objeto *options* permite definir um conjunto de parâmetros usados durante o teste. Neste caso, apenas é definido o número de iterações, ou seja, o número de vezes em que o teste é executado, e, caso não presente, executa uma única vez. Na verificação do resultado da execução, são obtidas três métricas:

1. código HTTP da resposta, sendo verificado se é igual a 200 (*status is 200* no código), indicando uma operação com sucesso;
2. duração do pedido, sendo feita uma verificação de que a duração é efetivamente sempre igual ou maior a 0 (*duration is >= 0* no código);
3. e o número de vezes em que pelo menos uma das duas verificações anteriores não foi verdadeira (*failureRate* no código).

No final de cada teste, espera-se que a percentagem das verificações 1 e 2 seja 100%, e a percentagem da verificação 3 seja 0%.

Listagem 5.2: *Script* executado pelo k6, para a obtenção dos tempos de execução de várias operações ao sistema.

```
export let options = {
    iterations: __ENV["ITERATIONS"] == null ? 1 : __ENV["ITERATIONS"],
};

const failureRate = new Rate("failure_rate");
const url = `http://${
        __ENV["HOST_ADDRESS"] == null ? "localhost" : __ENV["HOST_ADDRESS"]
    }:8080/api/${__ENV["URL"]}`;
const params = {
    headers: {
        "Authorization": `Basic ${encoding.b64encode("admin:admin")}`,
        "Content-Type": "application/json"
    },
    timeout: "300s"
};

export default function() {
    const method = __ENV["METHOD"].toUpperCase();
```

[17] Ferramenta de testes k6: https://k6.io

```
19      const requestBody = __ENV["REQUEST_BODY"];
20      let response = http.request(method, url, requestBody, params);
21      let checkRes = check(response, {
22          "status is 200": (r) => r.status === 200,
23          "duration is >= 0": (r) => r.timings.duration >= 0,
24      });
25      failureRate.add(!checkRes);
26  }
```

Os resultados dos tempos de execução, visíveis na tabela 5.1, foram obtidos usando a média de tempos de três execuções diferentes. De notar que os tempos de execução foram obtidos com uma velocidade de internet de aproximadamente 20 Mb/segundo de transferência e 30 Mb/segundo de carregamento, fator que influencia a velocidade de transferência de imagens *docker*, necessárias para iniciar os componentes do sistema: agente kafka, gestor local, servidor de registo e balanceador de carga, e para iniciar os contentores com serviços de aplicações. Outro fator que influencia o tempo de execução são as especificações da máquina onde é executado o gestor principal. Para tal, foi usado um portátil OMEN by HP 15-dc0xxx (4ML12EAAB9), com um CPU Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz com 12 *cores*, e 16 GB de memória RAM.

Examinando os valores dos tempos de execução, podemos chegar à conclusão que os componentes e os contentores de serviços demoram um tempo de aproximadamente 3 a 4 segundos a iniciar quando a imagem *docker* está em *cache*, ou seja, foi transferida previamente. Sendo que o tempo de iniciação quando a imagem não se encontra em *cache* depende do tamanho da mesma. O tempo maior de iniciação de um agente kafka deve-se ao facto de que para além da iniciação do contentor *docker*, também é preciso configurar os tópicos e iniciar os processos dos consumidores e produtores. Adicionalmente, também é possível verificar que a implementação de operações paralelas permitiu melhorar significativamente os tempos de execução das operações suportadas, referidas anteriormente na secção 4.2.8.3. No caso da obtenção das instâncias virtuais nas várias regiões AWS suportadas, o melhoramento é de quase 10 vezes menor tempo de execução. Já no caso da obtenção da lista de servidores registados nos balanceadores de carga, o melhoramento é de aproximadamente 5 vezes, quando são consideradas as regiões Europa, América do Norte, América do Sul, Ásia e Oceania.

5.4.2 Testes funcionais

Os testes funcionais garantem que as funcionalidades implementadas pelos componentes que compõem o sistema estão a funcionar corretamente. São apresentados vários testes a partes fundamentais do sistema, incluindo, quando pertinente, os tempos de execução dos vários passos necessários para implementar as funcionalidades.

Tabela 5.1: Tempos de execução de várias operações do sistema.

Operação	Tempo (s)
Iniciar e configurar uma instância virtual na AWS [1]	79,0
Terminar uma instância virtual na AWS	11,3
Iniciar um agente kafka (e respetivo zookeeper) (sem *cache*) [2]	214,0
Iniciar um agente kafka (e respetivo zookeeper) (com *cache*) [2]	11.9
Iniciar um gestor local (sem *cache*) [2]	197.0
Iniciar um gestor local (com *cache*) [2]	4.6
Iniciar um servidor de registo (sem *cache*) [2]	48,9
Iniciar um servidor de registo (com *cache*) [2]	2,7
Iniciar um balanceador de carga (sem *cache*) [2]	36,0
Iniciar um balanceador de carga (com *cache*) [2]	3,6
Iniciar um contentor do serviço *sock-shop-catalogue* (sem *cache*) [2]	31,1
Iniciar um contentor do serviço *sock-shop-catalogue* (com *cache*) [2]	3,6
Iniciar um contentor do serviço *hotel-reservation-frontend* (sem *cache*) [2]	11,26
Iniciar um contentor do serviço *hotel-reservation-frontend* (com *cache*) [2]	3,1
Obter as instâncias virtuais a executar na AWS *(sem paralelização)* [3]	5,72
Obter as instâncias virtuais a executar na AWS *(com paralelização)* [3]	0,59
Obter a lista de servidores dos balanceadores de carga *(sem paralelização)* [4]	3,4
Obter a lista de servidores dos balanceadores de carga *(com paralelização)* [4]	0,71
Sincronizar as instâncias na AWS com os dados no gestor principal [3]	0,65
Sincronizar os contentores dos *docker swarms* com os dados no gestor principal [5]	1,75
Sincronizar os nós dos *docker swarms* com os dados no gestor principal [5]	0,62

1. A configuração implica a entrada no *docker swarm*, incluindo o lançamento dos contentores de apoio: monitor de pedidos, *proxy* de autenticação e *prometheus*.
2. Tamanho de imagens comprimidas de contentores *docker*:
Monitor de pedidos - 7 MB. Proxy de autenticação - 9 MB. Prometheus - 63 MB.
Agente kafka - 196 MB. Zookeeper - 240 MB.
Gestor local - 338 MB. Servidor de registo - 99 MB. Balanceador de carga - 51 MB.
3. Zonas AWS consideradas: US East (N. Virginia), US East (Ohio), US West (N. California), US West (Oregon), Africa (Cape Town), Asia Pacific (Mumbai), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Tokyo), Canada (Central), Europe (Frankfurt), Europe (Ireland), Europe (London), Europe (Paris), Middle East (Bahrain), South America (São Paulo).
4. Regiões consideradas: Europa, América do Norte, América do Sul, Ásia, Oceania.
5. Considerando dois gestores locais na América do Norte e Ásia. Incluindo nós e contentores nas regiões América do Norte, Ásia e Europa. O gestor principal está localizado em Portugal.

5.4.2.1 Recolha de dados sobre pedidos de descoberta de serviços e de métricas sobre os sistemas operativos e contentores

Uma das funcionalidades fundamentais dos gestores é a capacidade de obter a informação sobre os pedidos de descoberta de serviços por parte de serviços dependentes. Outra característica de um gestor é a recolha de métricas sobre o sistema operativo dos nós e sobre os contentores que gere. Os dados obtidos são usados para se decidir qual é a melhor

localização aproximada onde se devem colocar os contentores dos serviços aplicacionais.

Neste teste são considerados vários nós a executar em diferentes regiões do sistema, e é verificado qual é o tempo de voo dos pedidos para se obterem os dados sobre os pedidos de descoberta de serviços por parte de serviços dependentes, bem como as métricas do sistema operativo e contentores em cada nó.

Para o teste, primeiro foram lançados dois nós na América do Norte, dois nós na Europa e dois nós na Ásia, e foram todos associados ao gestor principal, de forma a poderem ser obtidos os tempos de voo dos pedidos a partir do gestor principal. Depois, os nós das regiões América do Norte e Ásia foram associados ao respetivo gestor local de cada região, tendo sido obtidos os tempos de voo dos pedidos a partir de cada gestor local. A configuração do sistema durante este teste pode ser visualizada na figura 5.31.

Figura 5.31: Configuração do sistema durante o teste de recolha de dados.

Um das vantagens de serem usados gestores locais, é a sua maior proximidade aos recursos que cada um gere, quando comparado ao gestor principal. Portanto, é feita essa comparação na tabela 5.2. De notar que o gestor principal encontra-se a executar em Portugal.

Como é possível verificar, as médias dos tempos de voos dos pedidos para a recolha dos dados são mais baixas a partir do gestor da região onde se encontram os nós em questão. O que era de esperar, visto que a distância é melhor entre os recursos geridos por um gestor local, comparativamente com a distância entre os recursos do gestor principal, onde os recursos estão mais dispersos geograficamente.

5.4.2.2 Iniciação e configuração de uma instância virtual na *cloud*

Durante a execução dos gestores, caso necessitem de mais poder de processamento, ou caso decidam que a *cloud* é necessária para alojar o contentor de um serviço, pode ser preciso iniciar e configurar uma nova instância virtual na *cloud*. A ação tem vários passos, que podem ser medidos individualmente quanto ao seu desempenho:

1. É preciso enviar um pedido à AWS para iniciar o processo que inicia uma instância virtual nova.

Tabela 5.2: Tempos de voo (em milissegundos) dos pedidos para se obterem os dados sobre as descobertas de serviços por parte de serviços dependentes, bem como as métricas do sistema operativo e contentores, de cada nó, a partir de vários gestores.

Recolha de dados	Principal	Ásia	América
Descoberta de serviços de *Asia Pacific (Singapore)*	380	274	-
Métricas de *Asia Pacific (Singapore)*	954	698	-
Descoberta de serviços de *Asia Pacific (Seul)*	607	273	-
Métricas de *Asia Pacific (Seul)*	1536	698	-
Média	**869**	**483**	**-**
Descoberta de serviços de *US west (Oregon)*	358	-	133
Métricas de *US west (Oregon)*	912	-	362
Descoberta de serviços de *Canada (Montereal)*	250	-	30
Métricas de *Canada (Montereal)*	636	-	98
Média	**539**	**-**	**155**
Descoberta de serviços de *Europe (Paris)*	105	-	-
Métricas de *Europe (Paris)*	274	-	-
Descoberta de serviços de *Europe (London)*	98	-	-
Métricas de *Europe (London)*	261	-	-

2. É preciso esperar que a instância tenha iniciado completamente, e esteja pronta a receber comandos SSH.
3. De seguida é preciso iniciar o *proxy* de autenticação *docker*, usado para proteger o *docker* API.
4. Depois é iniciado um nó com o *role* WORKER, para se juntar ao *docker swarm* do gestor que controla a instância.
5. De seguida é preciso iniciar os componentes de apoio, que estão presentes em todos os nós do sistema, nomeadamente, o monitor de pedidos, e o *prometheus*. Os componentes são lançados em paralelo, portanto, o tempo de execução é calculado após ambos os componentes terem iniciado.
6. Por fim, é iniciado o processo do *node exporter*, para ser possível obter métricas sobre o sistema operativo *UNIX*.

5.4.2.3 Lançamento de um gestor local

O lançamento de um gestor local é uma das operações mais demorada do sistema. Não só devido ao tamanho da imagem *docker* (338 MB) de um gestor local, mas também pelo tempo que este demora a iniciar, receber todos os dados necessários do gestor principal e ficar pronto a gerir um conjunto de nós e contentores de serviços.

Neste teste, é iniciado um gestor local na região Ásia. É assumido que já existe um nó na região com capacidade para executar o contentor do gestor. Se não fosse o caso, era iniciada uma instância virtual numa zona AWS dentro da região, com o tempo de execução semelhante ao verificado no teste 5.4.2.2. Também é assumido que já existe um

Tabela 5.3: Tempos de execução (em segundos) dos passos para a iniciação e configuração de uma instância virtual na *cloud*.

Operação	Execução 1	Execução 2	Execução 3	Média
Pedido para iniciar a instância virtual na AWS.	1,41	1,35	1,40	1,39
Esperar que a instância virtual fique disponível.	47,1	46,3	45,0	46,1
Iniciar o componente proxy de autenticação *docker*.	20,0	21,3	20,5	20,6
Juntar o nó da nova instância virtual ao *docker swarm*.	3,0	2,9	3,3	3,1
Iniciar os componentes monitor de pedidos e prometheus (em paralelo).	4,6	4,3	4,1	4,3
Iniciar o processo do *node exporter*.	0,76	0,69	0,72	0,72
Total	**76,87**	**76,84**	**75,02**	**76,24**

agente kafka pronto na região considerada. Se tal não fosse o caso, seria necessário iniciar o componente, com um tempo de execução semelhante ao verificado na tabela de tempos de execução 5.1. Os passos para iniciar um gestor local na região Ásia, a partir do hub de gestão a executar em Portugal, foram repetidos três vezes, para se obterem valores mais exatos, os quais podem ser vistos na tabela 5.4.

Tabela 5.4: Tempos de execução (em segundos) dos passos para lançar um gestor local na Ásia.

Operação	Execução 1	Execução 2	Execução 3	Média
Iniciar o contentor *docker*.	40,1	38,2	40,4	39,6
Tempo até o gestor local enviar os primeiros dados sobre os nós e contentores que controla, indicando que iniciou comple--tamente.	33,3	35,3	34,5	34,3
Total	**73,4**	**73,5**	**74,9**	**73,9**

5.4.2.4 Lançamento de uma aplicação

O lançamento de uma aplicação é uma operação relativamente complexa, que permite lançar todos os serviços de uma aplicação em nós escolhidos pelo gestor. Neste teste é feito o lançamento da aplicação *Sock shop*, e é comparado o estado do sistema antes e depois da operação. Neste caso, o lançamento da aplicação é feito por um gestor local, que escolhe os locais mais adequados para iniciar os serviços da aplicação. Sendo que, como não existem máquinas na periferia da rede disponíveis, foi utilizada a *cloud* para alocar poder computacional *on-demand*.

Na configuração inicial do sistema, disponível na figura 5.32, existem três nós. Um nó em Portugal a executar o gestor principal. Outro nó na Alemanha, com um gestor local. E outro nó, também na Alemanha, com um agente *kafka* para a comunicação entre os gestores, com um servidor de registo para registar os serviços iniciados, e com um balanceador de carga para gerir a carga dos serviços *frontend*.

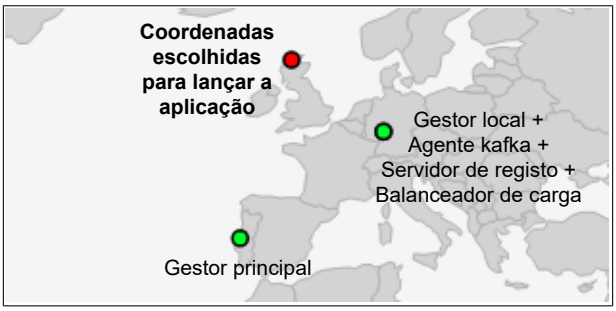


Figura 5.32: Configuração do sistema para o teste do lançamento de uma aplicação.

Durante o lançamento dos serviços da aplicação *Sock shop*, foram iniciadas, pelo gestor local, cinco instâncias virtuais em Londres, visto que era a localização mais perto das coordenadas selecionadas, com poder computacional disponível. A configuração do sistema após o teste pode ser vista na figura 5.33, cuja informação é relativa ao ponto localizado em Londres. É possível ver que os contentores foram iniciados em múltiplas máquinas virtuais, visto que o poder computacional de uma máquina virtual t2.micro só tem 1 computador virtual (1 Virtual Centralized Processing Unit (vCPU)) e 0,5 GB de RAM, sendo apenas suficiente para alojar um conjunto limitado de contentores.

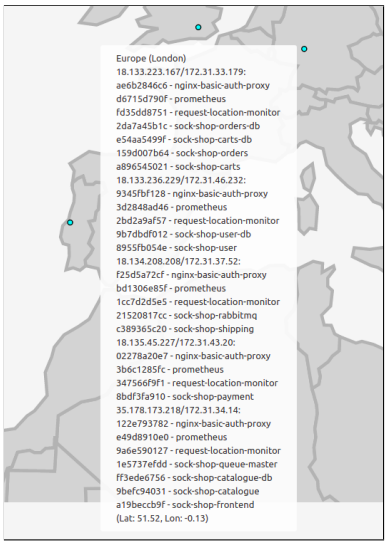


Figura 5.33: Configuração do sistema após o lançamento da aplicação *Sock shop* perto das coordenadas selecionadas.

5.4.2.5 Utilização do cliente e servidor de registo e descoberta de serviços

Para que seja possível a descoberta de serviços externos por parte de serviços dependentes, existe interação entre serviço ↔ cliente de registo e descoberta ↔ servidor de registo e descoberta ↔ monitor de pedidos. Neste teste é verificado como e quando é feita essa interação, e quais os dados que são necessários e que ficam guardados em cada componente. Para tal, o sistema é configurado como mostra a figura 5.34, com um gestor principal em Portugal, um servidor de registo e um balanceador de carga nos Estados Unidos, e são lançados três serviços, também nos Estados Unidos: *frontend*, *carts* e *catalogue* da aplicação *Sock shop*, sendo que existe uma réplica do serviço *carts* e duas réplicas do serviço *catalogue*. Como é possível ver na imagem 5.34, o serviço *frontend* encontra-se na máquina 54.243.169.71 (US East N. Virgínia), o serviço *catalogue* tem uma réplica na máquina 99.79.63.34 (Canada Montreal) e outra réplica na máquina 54.185.228.55 (US West Oregon), e a réplica do serviço *carts* encontra-se na máquina 54.185.228.55 (US West Oregon). O valor junto a cada ponto diz respeito ao identificador de cada nó do *docker swarm* que o gestor principal controla.

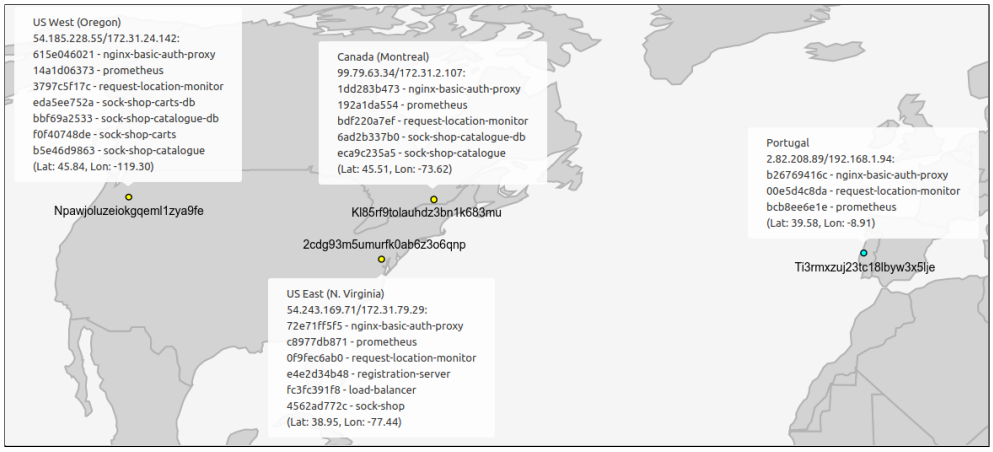


Figura 5.34: Configuração do sistema durante o teste ao cliente e servidor de registo e descoberta de serviços.

Ao aceder à página principal do serviço *frontend* da aplicação *Sock shop*, esta depende diretamente de dois serviços externos: *carts* e *catalogue*, ambos também da aplicação *Sock shop*. Portanto, para descobrir onde pode aceder a cada um dos serviços externos, o serviço *frontend* consulta o cliente de registo e descoberta de serviços duas vezes, através de pedidos REST.

O servidor de registo contém a informação sobre todas as réplicas registadas no sistema, visível na listagem 5.3. A informação sobre as réplicas registadas é obtida pelo cliente, que depois escolhe qual das réplicas deve devolver na resposta do pedido da descoberta de serviço, considerando os valores da latência e longitude das réplicas e do serviço dependente.

Listagem 5.3: Informação parcial que é guardada no servidor de registo e descoberta de serviços.

```
<applications>
  <application>
    <name>SOCK-SHOP-CATALOGUE</name>
    <instance>
      <instanceId>4d81dbbdd7b467391ecd36c5866049977fc263be</instanceId>
      <app>SOCK-SHOP-CATALOGUE</app>
      <ipAddr>99.79.63.34</ipAddr>
      <port enabled="true">8083</port>
      <status>UP</status>
      <vipAddress>sock-shop-catalogue</vipAddress>
      <metadata>
        <latitude>45.508968</latitude>
        <longitude>-73.616289</longitude>
      </metadata>
    </instance>
    <instance>
      <instanceId>d868b7fdb197da8a99352c46bb977324954a0531</instanceId>
      <app>SOCK-SHOP-CATALOGUE</app>
      <ipAddr>54.185.228.55</ipAddr>
      <port enabled="true">8083</port>
      <status>UP</status>
      <vipAddress>sock-shop-catalogue</vipAddress>
      <metadata>
        <latitude>45.841904</latitude>
        <longitude>-119.296774</longitude>
      </metadata>
    </instance>
  </application>
</applications>

<applications>
  <application>
    <name>SOCK-SHOP-CARTS</name>
    <instance>
      <instanceId>64853f8001e624d58c0c5c114d7cc7f5bb642e12</instanceId>
      <app>SOCK-SHOP-CARTS</app>
      <ipAddr>54.185.228.55</ipAddr>
      <port enabled="true">8085</port>
      <status>UP</status>
      <vipAddress>sock-shop-carts</vipAddress>
      <metadata>
        <latitude>45.841904</latitude>
        <longitude>-119.296774</longitude>
      </metadata>
    </instance>
  </application>
</applications>
```

Ao consultar os registos do contentor do serviço *frontend*, visível na listagem 5.5, é possível observar a interação que é feita entre o serviço, o cliente e servidor de registo e descoberta de serviços, e o monitor de pedidos.

Listagem 5.4: Registos do contentor do serviço *frontend* da aplicação *Sock shop*.

```
GET http://localhost:1906/api/services/sock-shop-catalogue/endpoint
Cached instances for service sock-shop-catalogue: []
Getting instances for VIP address "sock-shop-catalogue" from URL
    ↪ http://54.243.169.71:8761/eureka/vips/sock-shop-catalogue
GET http://localhost:1906/api/services/sock-shop-carts/endpoint
Got eureka response from url=http://54.243.169.71:8761/eureka/vips/sock-shop-catalogue
Instance chosen for sock-shop-catalogue: id = 4d81dbbdd7b467391ecd36c5866049977fc263be,
    ↪ address = 99.79.63.34:8083
Response from http://localhost:1906/api/services/sock-shop-catalogue/endpoint:
    ↪ 99.79.63.34:8083
Cached instances sock-shop-carts: []
Getting instances for VIP address "sock-shop-carts" from URL
    ↪ http://54.243.169.71:8761/eureka/vips/sock-shop-carts
Got eureka response from url=http://54.243.169.71:8761/eureka/vips/sock-shop-carts
Instance chosen for sock-shop-carts: id = 64853f8001e624d58c0c5c114d7cc7f5bb642e12,
    ↪ address = 54.185.228.55:8085
Response from http://localhost:1906/api/services/sock-shop-carts/endpoint:
    ↪ 54.185.228.55:8085
Sent location data {"service":"sock-shop-carts","count":1}
Sent location data {"service":"sock-shop-catalogue","count":1}
```

Após ser aplicado o algoritmo de descoberta de serviços, podemos verificar que a réplica do serviço *catalogue* escolhida estava localizada no Canada (Montreal), não sendo escolhida a réplica localizada em US West (Oregon), que estava mais longe do serviço *frontend*. Em relação ao serviço *carts*, a única réplica existente estava localizada em US West (Oregon), e portanto foi essa a réplica escolhida.

Os dados com as escolhas do algoritmo de descoberta de serviços (listagem 5.5) são enviados periodicamente pelo cliente de descoberta para o monitor de pedidos. Essa informação fica depois disponível aos gestores para ser usada nos algoritmos de migração e replicação de contentores. É possível ver que o nó "2cdg93m5umurfk0ab6z3o6qnp", onde se encontra o serviço *frontend*, requisitou uma vez a localização do serviço *carts*, e uma vez a localização do serviço *catalogue*, da aplicação *Sock shop*. Todos os outros três nós não requisitaram qualquer descoberta, portanto não existe informação disponível sobre os mesmos.

Listagem 5.5: Informação sobre os pedidos de descoberta de serviços externos, disponível nos gestores.

```
{
    "2cdg93m5umurfk0ab6z3o6qnp": {
        "sock-shop-carts": 1,
        "sock-shop-catalogue": 1
    },
```

```
    "kl85rf9tolauhdz3bn1k683mu": {},
    "ti3rmxzuj23tc18lbyw3x5lje": {},
    "npawjoluzeiokgqeml1zya9fe": {}
}
```

5.4.2.6 Teste à distribuição de carga de um serviço *frontend*

Neste teste é verificado como é feito a distribuição de pedidos de várias réplicas do serviço *frontend* da aplicação *Hotel reservation*. Para tal, é lançado um balanceador de carga na Europa e são lançadas quatro réplicas do serviço *frontend* distribuídas por nós na mesma região. A configuração do teste pode ser vista na figura 5.35. De notar que a réplica 4 está localizada num nó de uma máquina virtual AWS t2.medium, que tem 2 vCPUs e 4 GB de RAM, comparado com as outras três réplicas que estão em nós de máquinas virtuais AWS t2.micro, com 1 vCPU e 0,5 GB de RAM.

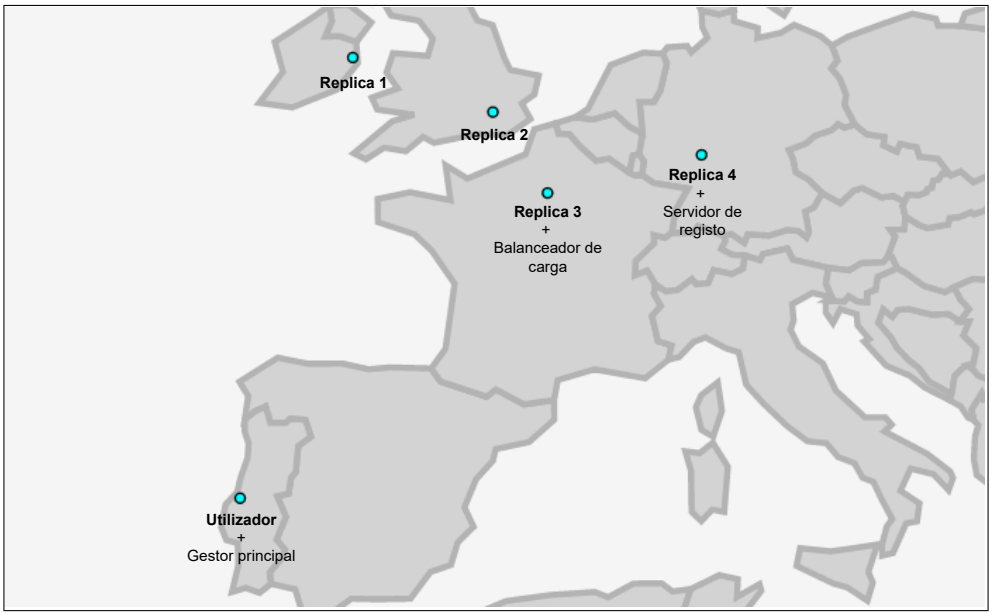


Figura 5.35: Configuração do teste de balanceamento de carga.

A estratégia de balanceamento usada é a *least connections*, o que significa que o balanceador de carga atribui o pedido à réplica com menos conexões ativas. Após 10000 pedidos por parte de 10 utilizadores em simultâneo, é obtido o gráfico da figura 5.36. O número de atribuições a cada réplica pode ser relacionado com a distância do balanceador de carga a cada nó. Como um pedido do balanceador de carga a uma réplica mais perto tem um tempo de voo menor, a réplica fica disponível mais rapidamente para receber outro pedido, comparativamente a outras réplicas mais longe. A réplica 4 (Alemanha) teve quase o mesmo número de pedidos da réplica 2 (Londres), apesar de estar um pouco mais longe do balanceador de carga, o que pode ser justificado pelo facto de que a réplica

4 está a executar numa máquina virtual com maior capacidade (t2.medium - 2 vCPU e 4 GB RAM) do que a máquina virtual a alojar a réplica 2 (t2.micro - 1 vCPU e 0,5 GB RAM).

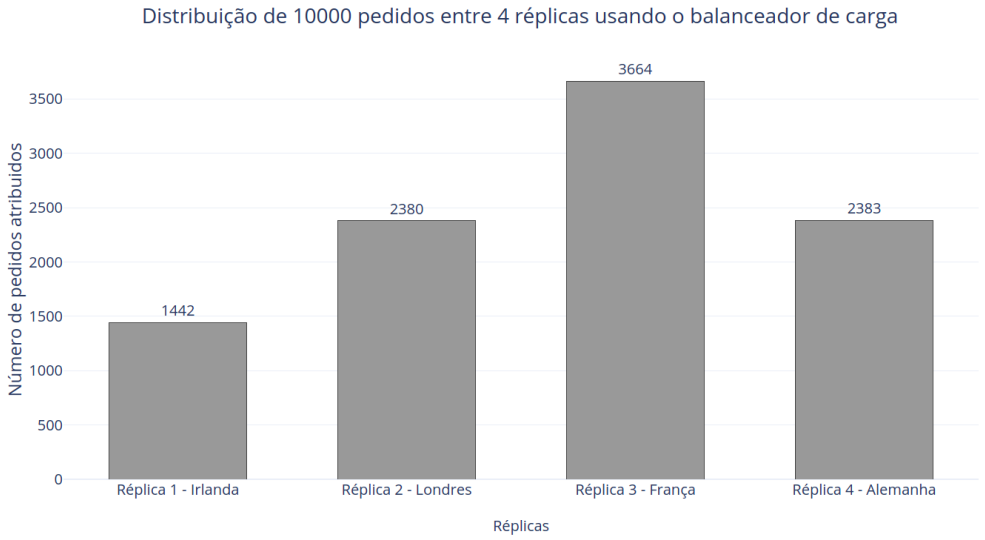


Figura 5.36: Resultado do teste de balanceamento de carga.

5.4.3 Testes de carga

Os testes de carga foram realizados através de vários cenários desenhados para demonstrar as capacidades do sistema, e avaliar o seu funcionamento e desempenho. Para o efeito, foram usadas duas aplicações: *Sock Shop* e *Hotel Reservation*, anteriormente referidas nos casos de estudo 5.3.

Para realizar os testes de carga, foi usada a ferramenta de testes k6 [18]. Após o desenvolvimento de *scripts* de testes, a ferramenta k6 permite criar utilizadores virtuais, sendo o seu número configurável através de parâmetros, que enviam pedidos HTTP ao *endpoint* desejado. Ao longo da execução do teste, a ferramenta k6 obtém várias métricas sobre o sucesso das operações, tempos de execução, número de iterações, quantidade de dados recebidos e enviados, e o número de utilizadores virtuais criados. Outra funcionalidade útil do k6 é o facto de permitir variar o número de utilizadores virtuais ao longo da execução do teste. Por exemplo, podem ser criados 50 utilizadores virtuais no primeiro minuto, depois 10 utilizadores virtuais nos 3 minutos seguintes, e por fim novamente 50 utilizadores virtuais no último minuto. A variabilidade de utilizadores permite simular o comportamento de um sistema adaptável às condições da rede e carga, como é o caso do protótipo desenvolvido nesta dissertação. Espera-se ver um resultado onde o número de réplicas de contentores vai diminuindo ou aumentando conforme o número de utilizadores virtuais criados pelo k6.

[18] Ferramenta de testes k6: https://k6.io/

Nas secções seguintes, são apresentados vários cenários que foram feitos para obter valores relevantes para a perceção do desempenho do sistema, e simular utilizadores de serviços lançados pelo sistema, com o objetivo de observar o comportamento do sistema.

5.4.3.1 Cenário 1 - Teste à latência

Um dos objetivos da computação na periferia da rede é diminuir a latência percebida pelos seus utilizadores. Portanto, é de esperar que a latência obtida ao aceder a um serviço a executar num nó na periferia da rede, perto do utilizador, seja menor à latência obtida no mesmo serviço a executar numa instância virtual na *cloud*, mais longe do utilizador. Este cenário considera duas réplicas do mesmo serviço a executarem em diferentes nós: uma réplica na *cloud*, outra na periferia da rede. O objetivo é comprovar que a latência é menor ao aceder à réplica na periferia da rede, comparado à latência obtida ao aceder à réplica na *cloud*.

A réplica a executar na *cloud* foi iniciada em Paris, na França, por ser a zona *cloud* AWS mais perto do utilizador, e a réplica a executar na periferia da rede está situada num nó de um computador pessoal, localizado a cerca de 5 km de distância do utilizador, como mostra o mapa da figura 5.37.

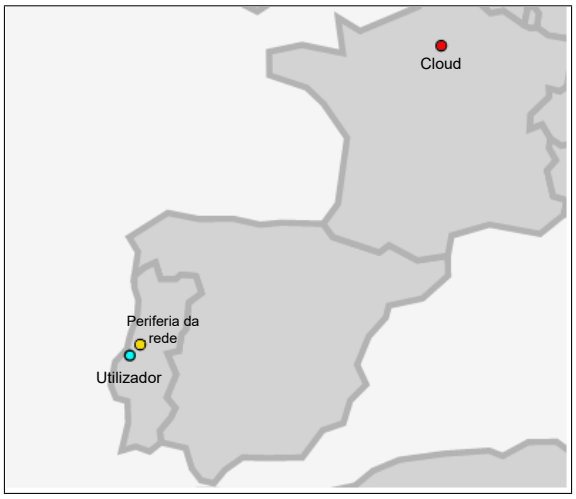


Figura 5.37: Mapa com a localização das réplicas do serviço *catalogue* da aplicação *Sock Shop*.

A figura 5.38 mostra os resultados obtidos após serem feitos 10000 pedidos às duas réplicas, por 10 utilizadores em simultâneo. A métrica mais relevante é o *http_req_duration*, que contém valores relacionados com o tempo total de cada pedido. Como era de esperar, a duração média do pedido (53,8 ms) à réplica a executar na *cloud*, mais longe do utilizador, é superior à duração média do pedido (28,8 ms) à réplica localizada na periferia da rede, mais perto do utilizador. Os valores de p(90) = 58,8 ms e p(95) = 59,66 ms também foram superiores no teste feito à réplica na *cloud*, comparativamente com a réplica na

periferia da rede, com valores p(90) = 37,9 ms e p(95) = 44,7 ms, o que indica que a maior parte dos pedidos à réplica na periferia da rede tiverem uma duração menor.

```
✓ status is 200

checks.....................: 100.00% ✓ 10000 ✗ 0
data_received..............: 32 MB   576 kB/s
data_sent..................: 930 kB  17 kB/s
http_req_blocked...........: avg=62.88µs  min=1.13µs  med=6.69µs  max=63.38ms  p(90)=9.12µs  p(95)=10.5µs
http_req_connecting........: avg=55.54µs  min=0s      med=0s      max=63.33ms  p(90)=0s      p(95)=0s
http_req_duration..........: avg=54.71ms  min=50.93ms med=54ms    max=83.75ms  p(90)=58.17ms p(95)=59.66ms
http_req_receiving.........: avg=867.39µs min=21.22µs med=620.3µs max=14.19ms  p(90)=1.97ms  p(95)=2.58ms
http_req_sending...........: avg=38.75µs  min=5.87µs  med=37.53µs max=1.47ms   p(90)=51.65µs p(95)=57.38µs
http_req_tls_handshaking...: avg=0s       min=0s      med=0s      max=0s       p(90)=0s      p(95)=0s
http_req_waiting...........: avg=53.8ms   min=50.04ms med=53.12ms max=83.6ms   p(90)=57.11ms p(95)=58.81ms
http_reqs..................: 10000   181.367815/s
iteration_duration.........: avg=55.05ms  min=51.23ms med=54.29ms max=136.25ms p(90)=58.44ms p(95)=59.96ms
iterations.................: 10000   181.367815/s
vus........................: 10      min=10  max=10
vus_max....................: 10      min=10  max=10
```

(a) Réplica na *cloud*.

```
✓ status is 200

checks.....................: 100.00% ✓ 10000  ✗ 0
data_received..............: 2.4 MB 80 kB/s
data_sent..................: 910 kB 30 kB/s
http_req_blocked...........: avg=50.74µs  min=0s      med=0s      max=30.22ms  p(90)=0s      p(95)=0s
http_req_connecting........: avg=22.28µs  min=0s      med=0s      max=28.38ms  p(90)=0s      p(95)=0s
http_req_duration..........: avg=28.83ms  min=10.87ms med=27ms    max=300.14ms p(90)=37.89ms p(95)=44.7ms
http_req_receiving.........: avg=609.51µs min=0s      med=0s      max=41.87ms  p(90)=1.11ms  p(95)=2.04ms
http_req_sending...........: avg=118.06µs min=0s      med=0s      max=33.94ms  p(90)=55.26µs p(95)=996.7µs
http_req_tls_handshaking...: avg=0s       min=0s      med=0s      max=0s       p(90)=0s      p(95)=0s
http_req_waiting...........: avg=28.11ms  min=10.87ms med=26.92ms max=300.14ms p(90)=36.9ms  p(95)=43.53ms
http_reqs..................: 10000  332.675863/s
iteration_duration.........: avg=29.95ms  min=10.95ms med=27.96ms max=300.14ms p(90)=39.17ms p(95)=46.04ms
iterations.................: 10000  332.675863/s
vus........................: 10     min=10 max=10
vus_max....................: 10     min=10 max=10
```

(b) Réplica na periferia da rede.

Figura 5.38: Métricas obtidas após 10000 pedidos, feitas por 10 utilizadores, a duas réplicas do serviço *catalogue* da aplicação *Sock Shop*, em localizações distintas.

5.4.3.2 Cenário 2 - Carga a um gestor local

Neste cenário são obtidas métricas sobre a capacidade de um gestor local, quanto aos recursos que consome, consoante o número de nós e contentores que gere. Para realizar o teste, foi lançado um gestor local na Europa, a executar numa máquina virtual na AWS, do tipo t2.medium, com 2 vCPU e 4 GB de RAM. As métricas do gestor local foram obtidas através de um programa programado em Java, que obtém as métricas, através do comando "docker stats" [19], do contentor com o identificador passado por parâmetro, a executar numa máquina com um certo endereço, também passado por parâmetro. As métricas obtidas são a percentagem de CPU usada, a memória RAM consumida, o número de *bytes* recebidos e o número de *bytes* enviados. Os valores das métricas são obtidos de 2 em 2 segundos, e são imediatamente escritos num ficheiro com formato Comma-separated values (CSV), para serem depois consultados e para facilitar a construção de gráficos.

O teste consiste em 2 minutos de iniciação, onde o gestor inicia e recebe os dados do gestor principal, nesta altura apenas tem um nó para gerir e nenhum contentor aplicacional. Depois são iniciadas e configuradas 10 instâncias virtuais por 2 utilizadores em

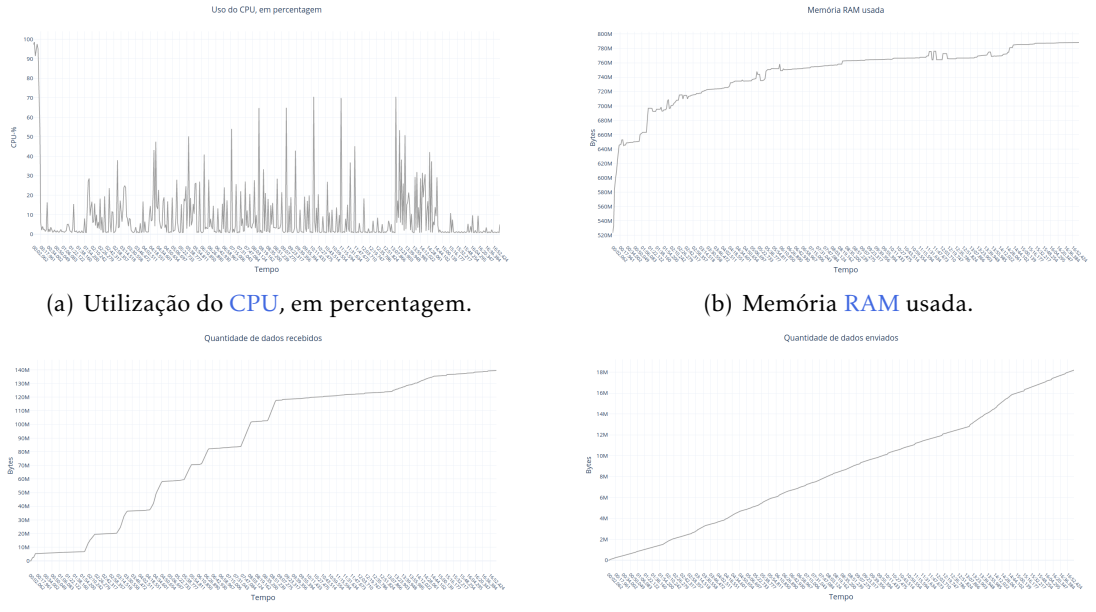
[19] Docker stats: https://docs.docker.com/engine/reference/commandline/stats

simultâneo, o que significa que o gestor local passa a gerir 11 nós, contando com o próprio nó onde executa o seu contentor. De seguida, existem 2 minutos de repouso, para se obterem métricas sobre a gestão dos nós. Depois, são iniciados, por 2 utilizadores, serviços *frontend* da aplicação *Hotel reservation*, até um total de 25 contentores. Por fim, são feitos mais 2 minutos de repouso de modo a serem obtidas métricas sobre a gestão dos nós e contentores.

O teste pode ser dividido em cinco fases:

1. Iniciação do gestor local (2 minutos);
2. Inicio e configuração de 10 instâncias virtuais na AWS, por parte de 2 utilizadores em simultâneo (9 minutos e 17,6 segundos);
3. Repouso de 2 minutos para se obterem métricas relativas à gestão dos nós iniciados na segunda fase (2 minutos);
4. Inicio de 25 contentores do serviço *frontend* da aplicação *Hotel reservation* (1 minuto e 39,9 segundos);
5. Repouso de 2 minutos para se obterem métricas relativas à gestão dos nós e contentores iniciados anteriormente (2 minutos);

O resultado do teste está disponível nos gráficos da figura 5.39.



(a) Utilização do CPU, em percentagem.

(b) Memória RAM usada.

(c) Quantidade de dados recebidos.

(d) Quantidade de dados enviados.

Figura 5.39: Resultado do teste de carga efetuado a um gestor local.

É possível observar que no inicio da execução do gestor, o consumo de CPU é elevado, o que pode estar associado ao processamento dos dados do *kafka*, que são enviados pelo gestor principal. Depois, o consumo de CPU estabiliza num valor baixo, de aproximadamente 1 a 5%. As instâncias virtuais são iniciadas e configuradas após o minuto 2, o que pode ser visto no gráfico 5.39(a) com um aumento significativo de consumo de CPU de

pouca duração, representado pelo pico visível no gráfico. Como existem 2 utilizadores virtuais, os 10 pedidos chegam paralelamente 2 a 2, e como foi visto anteriormente na secção 5.4.2.3, cada pedido tem uma duração aproximada de 75 segundos, o que implica que o aumento breve do consumo CPU acontece em intervalos de aproximadamente 75 em 75 segundos. Em relação à fase de iniciação dos contentores, que acontece por volta do minuto 13:15 até ao minuto 14:55, podemos ver que o valor médio da utilização do CPU foi mais elevado do que nas outras fases do teste. O que indica que a iniciação de contentores é uma operação mais pesada computacionalmente do que a iniciação e configuração de instâncias virtuais na *cloud*, o que faz sentido, visto que o processo de iniciação de instâncias virtuais é delegado ao serviço da AWS.

Quanto à memória RAM usada, o valor cresceu muito rápido na fase de iniciação do gestor, talvez devido à iniciação dos componentes necessários à *framework* Spring, incluindo a base de dados que é guardada em memória. Após o primeiro minuto, existe outro salto da quantidade de memória RAM usada, que pode estar associado ao primeiro ciclo de monitorização de nós e contentores, altura em que são também criadas as regras usadas nas decisões de migração e replicação de contentores, e paragem e inicio de nós. A partir deste momento, o uso de RAM estabiliza um pouco, apenas aumentando lentamente até ao final do teste.

O gráfico da quantidade de dados recebidos tem subidas mais acentuadas, de aproximadamente 75 em 75 segundos, coincidindo com o momento de iniciação das instâncias virtuais na AWS. Depois, por volta do minuto 13:15, é possível ver uma subida dos dados recebidos por segundo, o que coincide com a fase de iniciação dos contentores, indicando que o gestor local está a receber informação sobre os contentores iniciados.

Já o gráfico da quantidade de dados enviados representa uma função aproximadamente linear, com um decline mais acentuado na fase de iniciação dos contentores, a partir do minuto 13:15, o que pode ser causado pelo envio da informação sobre os contentores ao gestor principal. No gráfico, é também possível verificar pequenos declives mais acentuados que acontecem no inicio de cada minuto, o qual pode ser explicado pelo ciclo de monitorização de nós e contentores, e o respetivo envio das métricas obtidas e das decisões tomadas, ao gestor principal.

5.4.3.3 Cenário 3 - Sobrecarga a uma réplica de um serviço

O objetivo deste cenário é verificar se o sistema deteta o número excessivo de utilizadores a aceder a uma réplica de um serviço, e reage de acordo com as regras definidas e algoritmos desenvolvidos para a migração e replicação de contentores. Como exemplo, foi usado o serviço *frontend* da aplicação *Sock shop*. As regras definidas e aplicadas nas decisões do que fazer aos contentores foram:

1. Replicar o contentor quando o número de *bytes* enviados por segundo é superior a 100000.
2. Parar o contentor quando o número de *bytes* enviados por segundo é inferior a 1000.

De relembrar que uma ação de replicação só é executada quando se deteta que é preciso replicar o contentor em dois ciclos de monitorização consecutivos, e uma ação de paragem de um contentor apenas é feita quando se decide parar o contentor em três ciclos de monitorização consecutivos. Outra nota importante, é o facto de que o ciclo de monitorização acontece de 30 em 30 segundos, num gestor local.

No teste são considerados dois casos: quando a réplica é acedida por utilizadores apenas numa localização, nos Estados Unidos (Califórnia), e quando a réplica é acedida por dois grupos de utilizadores, um nos Estados Unidos (Califórnia) e outro grupo no Canadá (Montreal). O objetivo é verificar se o gestor deteta a sobrecarga à réplica, e replica o serviço na localização mais adequada, considerando a necessidade e localização dos utilizadores.

De notar que os utilizadores acedem ao balanceador de carga para encontrarem a réplica do serviço *frontend*, o que comprova o bom funcionamento da funcionalidade implementada no balanceador, que permite injetar a latitude e a longitude do utilizador no cabeçalho dos pedidos, usado a base de dados *MaxMind GeoIP2* [20] para encontrar a localização do utilizador.

Durante o teste, em ambos os casos, o gestor local está localizado nos Estados Unidos (Virgínia), o gestor principal encontra-se a executar em Portugal, e a réplica do serviço *frontend* da aplicação *Hotel reservation* está inicialmente localizada nos Estados Unidos (Oregon), como mostra o mapa da figura 5.40.

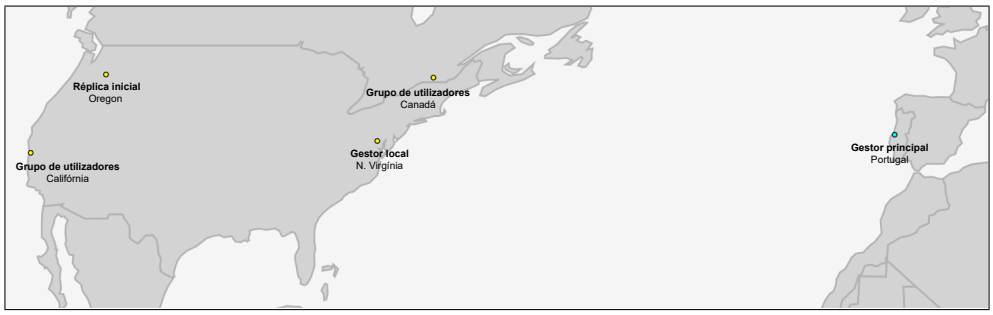


Figura 5.40: Mapa com a configuração inicial do cenário de sobrecarga a uma réplica de um serviço.

As opções do *script* k6 do teste usado para simular o segundo caso do cenário estão presentes na listagem 5.6. Para o primeiro caso, a diferença está na distribuição dos pedidos, que têm origem 100% em *Palo Alto*, cidade da Califórnia. Mas, devido a restrições da versão gratuita do k6, apenas foi possível usar um local para o envio dos pedidos. Por forma a simular o comportamento de múltiplos locais, foram executados dois testes em simultâneo, cada um no local pretendido. Como efeito, o número de pedidos com origem em cada local passou a não ser distribuído em 50%, mas foi distribuído consoante a capacidade da máquina e da rede de cada local. No final do teste, foi possível verificar que a distribuição foi de aproximadamente 75% para o Canadá e 25% para a Califórnia.

[20] MaxMind GeoIP2: https://www.maxmind.com/en/geoip2-databases

Mas o importante era existirem dois locais distintos com utilizadores a acederem ao serviço, o que permitiu chegar às conclusões necessárias.

O teste foi dividido em 5 fases:

1. 1 minuto sem utilizadores a acederem ao serviço.
2. 3 minutos com intensidade moderada, não devendo ser suficiente para sobrecarregar a réplica e causar uma replicação do contentor do serviço.
3. 3 minutos com intensidade elevada, de forma a sobrecarregar a réplica e, através da recolha das métricas e aplicação das regras e algoritmos de migração/replicação, causar uma ação de replicação do contentor do serviço.
4. 3 minutos com intensidade baixa, de forma a verificar se as eventuais réplicas iniciadas na fase anterior são paradas, ou continuam em execução.
5. 1 minuto sem utilizadores a acederem ao serviço.

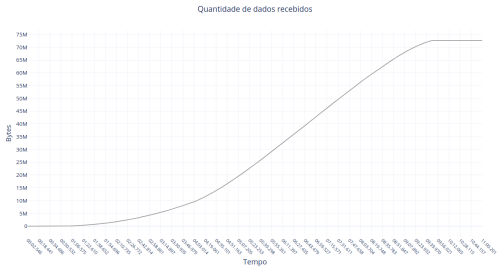
Listagem 5.6: *Script* k6 usado para simular o cenário com carga variável a uma réplica do serviço *frontend* da aplicação *Hotel reservation*.

```
export let options = {
    stages: [
        {target: 10, duration: "3m"},
        {target: 50, duration: "3m"},
        {target: 5, duration: "3m"},
    ],
    ext: {
        loadimpact: {
            distribution: {
                // 50% dos pedidos com origem em Palo Alto, California
                uspalo: {loadZone: "amazon:us:palo alto", percent: 50},
                // 50% dos pedidos com origem em Montreal, Canada
                montreal: {loadZone: "amazon:ca:montreal", percent: 50},
            }
        }
    }
}
```

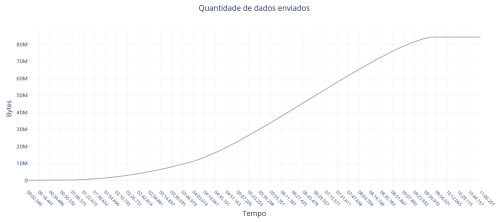
Durante o teste, as métricas do contentor da réplica inicial foram monitorizadas, usando o mesmo programa que foi usado no teste de carga ao gestor local. Foram obtidas métricas sobre o número total de *bytes* recebidos e enviados, e o número de *bytes* recebidos e enviados por segundo. Os gráficos da figura 5.41 mostram as métricas obtidas ao longo do decorrer do teste. É possível ver uma clara distinção no número de *bytes* recebidos e enviados durante as cinco fases do teste, à medida que o número de utilizadores é aumentado ou diminuído.

Acedido por um grupo de utilizadores na Califórnia

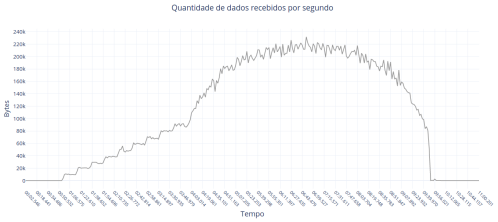
Neste caso foram feitos 65559 pedidos ao serviço *frontend* da aplicação *Sock shop* a partir da Califórnia, sendo que a localização dos utilizadores, obtida da base de dados *MaxMind*



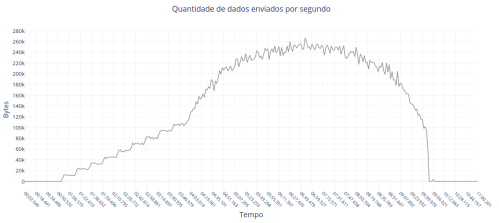
(a) Quantidade de bytes recebidos.



(b) Quantidade de bytes enviados.



(c) Quantidade de bytes recebidos por segundo.



(d) Quantidade de bytes enviados por segundo.

Figura 5.41: Métricas obtidas ao contentor durante o teste de carga.

GeoIP2, foi associada às coordenadas com latitude 37.18 e longitude -121.79, como é possível ver na figura 5.42. Por volta dos minutos 5:30, 6:30, 7:30 e 8:30, o gestor decidiu replicar o contentor inicial do serviço, o que implicou que o gestor replicá-se o contentor na zona AWS na Califórnia, visto que era a zona com poder computacional mais perto das coordenadas calculadas para o ponto médio dos acessos ao serviço. Quanto à regra de paragem do contentor, não houve tempo suficiente para ser acionada, visto que são precisos três ciclos de monitorização para que isso aconteça. No entanto, com o número de utilizadores reduzido após o minuto 10, a decisão iria ser entretanto começar a parar as réplicas iniciadas durante o teste de carga.


```
{
  "r92nmzbc47i1njjrchlvln67f": {},
  "rju1n1ndd6mllummd1658rzt0": {},
  "mz2ev71m2p1aqdut1cs5kdfts": {},
  "oxpun4g8doeniltcj1od0nnt5": {},
  "xllrid9y465xg9p84846hg3lp": {
    "sock-shop": [
      {
        "count": 65559,
        "latitude": 37.1807,
        "longitude": -121.787
      }
    ],
    "sock-shop-catalogue": [
      {
        "count": 65557,
        "latitude": 45.841904,
        "longitude": -119.296774
      }
    ]
  }
}
```


(a) Número de pedidos feitos aos serviços em cada nó do sistema.


```
{
  "sock-shop": [
    {
      "coordinates": {
        "label": null,
        "title": null,
        "latitude": 37.1807,
        "longitude": -121.787
      },
      "weight": 65559
    }
  ],
  "sock-shop-catalogue": [
    {
      "coordinates": {
        "label": null,
        "title": null,
        "latitude": 45.841904,
        "longitude": -119.296774
      },
      "weight": 65557
    }
  ]
}
```


(b) Cálculos do peso de cada localização, para cada serviço.


```
{
  "sock-shop": {
    "label": null,
    "title": null,
    "latitude": 37.18070000000001,
    "longitude": -121.787
  },
  "sock-shop-catalogue": {
    "label": null,
    "title": null,
    "latitude": 45.84190399999999,
    "longitude": -119.29677400000001
  }
}
```


(c) Calculo do ponto médio com peso, para cada serviço.

Figura 5.42: Informação sobre os pedidos dos utilizadores e de descoberta de serviços, em cada um dos nós do sistema, durante o primeiro caso do teste de carga.

Acedido por dois grupos de utilizadores, um na Califórnia, outro no Canadá

No caso em que o teste é feito com dois grupos de utilizadores, um na Califórnia, outro no Canadá, as métricas obtidas são semelhantes ao teste anterior. Mas, como a localização dos pedidos é diferente, os valores obtidos nos cálculos dos pesos e pontos médios, figura 5.43, são também diferentes. Agora, o ponto médio calculado para o serviço *frontend* da aplicação *Sock shop* foi na latitude 40.90 e na longitude -111.76, o que se traduz numa coordenada próxima de *Salt Lake City*, mais ou menos a meio das duas localizações onde originaram os pedidos, visto que o número de pedidos feito nos dois locais foi agora considerado no cálculo. Novamente, por volta dos minutos 5:30, 6:30, 7:30 e 8:30, o gestor verificou que a regra da replicação foi acionada, o que obrigou à replicação do serviço *frontend* para a zona com poder computacional mais perto do ponto médio calculado, que acabou por ser na cidade de *Oregon*, onde está disponível uma zona AWS. Em relação à regra de paragem do contentor, o tempo do teste não permitiu novamente que a decisão de parar um contentor fosse executada, no entanto, iria acontecer pouco tempo depois do teste ter terminado.

```
{
  "mlc921tmq9wwi957t20ehngyb": {},
  "ka73e0di0dfh8m7su7ezcerzi": {},
  "bgsjkyldexwowm5bvng0dcilz": {},
  "szbxg460w7tpz72f0ix4g2gg3": {},
  "n7bhtu2ut80vuqzg93k2y6j2a": {
    "sock-shop": [
      {
        "count": 31216
        "latitude": 45.4995,
        "longitude": -73.5848
      }
    ],
    "sock-shop-catalogue": [
      {
        "count": 31216
        "latitude": 45.841904,
        "longitude": -119.296774
      }
    ]
  },
  "f4mj6m9q5wi5kufnft8sc96ap": {
    "sock-shop": [
      {
        "count": 9992
        "latitude": 45.4995,
        "longitude": -73.5848
      }
    ],
    "sock-shop-catalogue": [
      {
        "count": 9992
        "latitude": 45.841904,
        "longitude": -119.296774
      }
    ]
  },
  "qeykcwk8l3cemt4rw96y7uuj3": {}
}
```

```
{
  "sock-shop": [
    {
      "coordinates": {
        "label": null,
        "title": null,
        "latitude": 37.1807,
        "longitude": -121.787
      },
      "weight": 31216
    },
    {
      "coordinates": {
        "label": null,
        "title": null,
        "latitude": 45.4995,
        "longitude": -73.5848
      },
      "weight": 9992
    }
  ],
  "sock-shop-catalogue": [
    {
      "coordinates": {
        "label": null,
        "title": null,
        "latitude": 45.841904,
        "longitude": -119.296774
      },
      "weight": 41493
    }
  ]
}
```

```
{
  "sock-shop": {
    "label": null,
    "title": null,
    "latitude": 40.908342541932356,
    "longitude": -111.76314516601613
  },
  "sock-shop-catalogue": {
    "label": null,
    "title": null,
    "latitude": 45.84190399999999,
    "longitude": -119.29677400000001
  }
}
```


(a) Número de pedidos feitos aos serviços em cada nó do sistema.

(b) Cálculos do peso de cada localização, para cada serviço.

(c) Cálculo do ponto médio com peso, para cada serviço.

Figura 5.43: Informação sobre os pedidos dos utilizadores e de descoberta de serviços, em cada um dos nós do sistema, durante o segundo caso do teste de carga.

Capítulo 6

Conclusões e trabalho futuro

Nesta dissertação foi apresentado um protótipo de uma arquitetura hierárquica de um sistema de gestão dinâmica de micro-serviços num ambiente que combina recursos heterogéneos no continuo *cloud*/*edge*. Através de uma hierarquia de gestores, foi possível definir regiões onde são iniciados e geridos um conjunto de nós e contentores. O número de nós e contentores pode ser distribuído por vários gestores locais, permitindo a distribuição de responsabilidades, e redução da região geográfica abrangida por cada gestor, reduzindo assim a distância de comunicação entre nós. Ao tirar partido de um conjunto de nós na periferia de rede, evita-se o envio de dados para a *cloud*, contribuindo para a diminuição dos dados transferidos na rede. Para mais, a computação na *cloud* na maior parte das vezes implica uma latência maior devido à distância aos utilizadores ser também maior, comparativamente à computação na periferia da rede. A latência menor da computação na periferia da rede permite que os serviços ofereçam melhor qualidade de serviço (QoS) percebida pelos utilizadores.

O protótipo apresentado pretende tirar partido das máquinas existentes na periferia da rede, para alojar serviços, através da gestão de nós e contentores com recurso à recolha e análise de métricas, aplicação de regras e execução de decisões. Das quais resultam a iniciação ou paragem de nós, e a migração e/ou replicação de contentores entre nós. Para a definição de aplicações, serviços, regras e métricas simuladas, e para executar ações e visualizar o progresso do sistema, foi desenvolvida uma aplicação *web* onde o utilizador pode aceder através de um *browser*. Para controlar o sistema como um todo, foi desenvolvido um gestor principal, usado para fazer a ligação entre todos os componentes do sistema, e que tem o conhecimento total do sistema. Para tirar partido total da informação sobre a localização das replicas dos serviços, foram desenvolvidos algoritmos de migração e replicação de contentores, baseados em distâncias com recurso a coordenadas.

Para testar as funcionalidades, para além da aplicação usada no trabalho anterior,

Sock Shop, foram incluídas aplicações compostas por micro-serviços, lançados dentro de contentores. Entre as aplicações estão: *Social Network*, *Media*, *Hotel Reservation*, *Online Boutique* e *Testing suite*, sendo que a *Hotel Reservation* foi a única que foi adaptada para usar a funcionalidade de descoberta de serviços incluída no sistema. O conjunto de aplicações permitiu confirmar a utilidade de um sistema de gestão automática de micro-serviços, ao reduzir a latência percebida pelos utilizadores.

Concluindo, embora nem todos os objetivos definidos inicialmente tivessem sido atingidos (mais métricas de decisão) outras características identificadas durante a implementação, como o desenvolvimento da aplicação *web*, ou a implementação de operações paralelas e/ou assíncronas, foram consideradas.

6.1 Trabalho futuro

No seguimento do trabalho feito nesta dissertação, sugiro algumas melhorias ao nível dos componentes do sistema, bem como na integração do trabalho das vertentes feitas noutras dissertações, onde esta dissertação está inserida:

1. **Suportar vários gestores locais por região**. Seria necessário controlar a sua posição dentro da região, distribuir a carga entre os gestores existentes na região, e lançar e parar gestores dinamicamente consoante a necessidade da região. Parte da implementação do gestor principal já assume que existem vários gestores locais, balanceadores de carga, servidores de registo e agentes kafka por região, para ser mais fácil a sua extensão.

2. **Inclusão da latência como métrica para decisão do local de migração dos contentores**. Atualmente o fator predominante para a decisão da localização de migração de um contentor é a distância geográfica e o número de pedidos, como foi descrito no algoritmo 1. A distância geográfica é um bom indicador, mas nem sempre é o fator que determina a latência. É preciso ter em consideração o tempo de transmissão e o tempo de processamento dos pedidos aos serviços.

3. **Suporte a mais provedores de computação *cloud***. Nesta dissertação o foco foi a extensão do sistema para várias zonas cloud, mas apenas suportadas pela Amazon. O próximo passo evidente é incluir outros fornecedores cloud, para evitar *lock-in* na Amazon, e ter disponível mais zonas de computação *on-demand*. Entre as possíveis escolhas estão o Microsoft Azure e o Google cloud.

4. **Deteção de previsão de acontecimentos autónoma**. Atualmente, a previsão de acontecimentos tem que ser especificada através do Hub de gestão, o que vai contra a corrente de execução do sistema, que procura automatizar o processo completo da gestão. Com a implementação de um mecanismo que consiga detetar acontecimentos, o sistema pode ser mais autónomo. Mas surgem questões relativas à quantidade

de informação necessária para tornar isso uma realidade. Acontecimentos que ocorrem de hora a hora, ou talvez diariamente, são mais fáceis de prever, mas outros acontecimentos mais raros ou aleatórios são difíceis de prever.

5. **Migração e replicação de contentores entre regiões**. Atualmente, o sistema isola cada região associado a um gestor local. Os micro-serviços com dependências a serviços localizados em regiões diferentes podem, eventualmente, migrar para regiões controlados por outro gestor local, mas a solução atual não suporta que isso aconteça.

6. **Aprimoramento dos algoritmos**. Quando é detetado que um nó tem carga a mais, é adicionado outro nó o mais próximo possível, e é migrado um contentor aleatório para o novo nó. A aleatoriedade não é ideal, porque a localização atual do contentor selecionado podia já ser ótima. O sistema pode beneficiar de um algoritmo que seja capaz de escolher melhor qual, ou quais, os contentores que devem ser migrados, para balancear a carga entre os nós.

7. **Integração dos componentes nas outras vertentes do sistema**: monitorização e gestão de dados, desenvolvidas no âmbito de outras dissertações.

Bibliografia

[1] Amazon. *Cloud Products*. URL: https://aws.amazon.com/products (acedido em 02/2019).

[2] G. Bierman, M. Abadi e M. Torgersen. «Understanding TypeScript». Em: *ECOOP 2014 – Object-Oriented Programming*. Ed. por R. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9.

[3] K. Bilal et al. «Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers». Em: *Computer Networks* 130 (2018), pp. 94–120. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1389128617303778.

[4] S. Carlini. *The Drivers and Benefits of Edge Computing*. Rel. téc. 226. Schneider Electric's Data Center Science Center, fev. de 2019. URL: https://www.schneider-electric.com/en/download/document/APC_VAVR-A4M867_EN/.

[5] A. V. Carrusca. «Gestão de micro-serviços na Cloud e Edge». Tese de mestrado. Largo da Torre, 2825-149 Caparica: Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, set. de 2018.

[6] N. Dragoni et al. «Microservices: Yesterday, Today, and Tomorrow». Em: *Present and Ulterior Software Engineering*. Ed. por M. Mazzara e B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[7] C. Esposito, A. Castiglione e K. R. Choo. «Challenges in Delivering Software in the Cloud as Microservices». Em: *IEEE Cloud Computing* 3.5 (set. de 2016), pp. 10–14. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.105.

[8] C.-F. Fan, A. Jindal e M. Gerndt. «Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application». Em: jan. de 2020, pp. 204–215. DOI: 10.5220/0009792702040215.

[9] N. C. Gabriel J.X. Dance Michael LaForgia. Dez. de 2018. URL: https://www.nytimes.com/2018/12/18/technology/facebook-privacy.html (acedido em 02/2019).

[10] Y. Gan et al. «An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud Edge Systems». Em: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[11] P. Garcia Lopez et al. «Edge-Centric Computing: Vision and Challenges». Em: *SIGCOMM Comput. Commun. Rev.* 45.5 (set. de 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: https://doi.org/10.1145/2831347.2831354.

[12] Google. *Products and services*. URL: https://cloud.google.com/products (acedido em 02/2019).

[13] C.-H. Hong e B. Varghese. «Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms». Em: *ACM Comput. Surv.* 52.5 (set. de 2019). ISSN: 0360-0300. DOI: 10.1145/3326066. URL: https://doi.org/10.1145/3326066.

[14] S. N.-T. Klervie Toczé. «Where Resources Meet at the Edge». Em: *2017 IEEE International Conference on Computer and Information Technology (CIT)*. Helsinki, Finland: IEEE, set. de 2017. ISBN: 978-1-5386-0958-3. DOI: https://doi.org/10.1109/CIT.2017.60. URL: https://ieeexplore.ieee.org/document/8031490.

[15] Kubernetes. URL: https://kubernetes.io (acedido em 02/2019).

[16] I. B. Lahmar e K. Boukadi. «Resource Allocation in Fog Computing: A Systematic Mapping Study». Em: *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2020, pp. 86–93. DOI: 10.1109/FMEC49853.2020.9144705.

[17] D. C. Marinescu. «Cloud Computing: Theory and Practice». Em: *Introduction*. First. Morgan Kaufmann, 2013. Cap. 1, pp. 1–21. ISBN: 978-0-12404-627-6.

[18] D. C. Marinescu. «Cloud Computing: Theory and Practice». Em: *Cloud Computing: Applications and Paradigms*. First. Morgan Kaufmann, 2013. Cap. 4, pp. 99–131. ISBN: 978-0-12404-627-6.

[19] D. C. Marinescu. «Cloud Computing: Theory and Practice». Em: *Cloud Resource Virtualization*. First. Morgan Kaufmann, 2013. Cap. 5, pp. 131–163. ISBN: 978-0-12404-627-6.

[20] J. L. Martin Fowler. *Microservices*. 25 de mar. de 2014. URL: https://martinfowler.com/articles/microservices.html (acedido em 02/2019).

[21] J. L. Martin Fowler. *Microservices*. URL: https://martinfowler.com/microservices (acedido em 02/2019).

[22] P. Mell e T. Grance. *The NIST Definition of Cloud Computing*. Rel. téc. 800-145. Computer Security Division, Information Technology Laboratory, National Institute of Standards e Technology, Gaithersburg, MD 20899-8930: National Institute of Standards e Technology, set. de 2011.

[23] D. Nazareth e J. Choi. «Market Share Strategies for Cloud Computing Providers». Em: *Journal of Computer Information Systems* (fev. de 2019), pp. 1–11. DOI: 10.1080/08874417.2019.1576022.

[24] R. B. Redowan Mahmud Ramamohanarao Kotagiri. «Fog Computing: A Taxonomy, Survey and Future Directions». Em: *Internet of Everything. Internet of Things (Technology, Communications and Computing)*. Out. de 2017, pp. 103–130. DOI: https://doi.org/10.1007/978-981-10-5861-5_5. URL: https://link.springer.com/chapter/10.1007%2F978-981-10-5861-5_5.

[25] M. Satyanarayanan. «The Emergence of Edge Computing». Em: *Computer*. Vol. 50. IEEE, jan. de 2017, pp. 30–39. DOI: https://doi.org/10.1109/MC.2017.9. URL: https://ieeexplore.ieee.org/document/7807196.

[26] A. Sill. «Standards at the Edge of the Cloud». Em: *IEEE Cloud Computing*. IEEE, abr. de 2017, pp. 63–67. DOI: https://doi.org/10.1109/MCC.2017.23. URL: https://ieeexplore.ieee.org/document/7912167.

[27] B. Varghese et al. «Challenges and Opportunities in Edge Computing». Em: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. Nov. de 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[28] M. Villari et al. «Osmotic Computing: A New Paradigm for Edge/Cloud Integration». Em: *IEEE Cloud Computing* 3.6 (nov. de 2016), pp. 76–83. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.124.

[29] G. A. Vitor Goncalves da Silva Marite Kirikova. «Containers for Virtualization: An Overview». Em: *Applied Computer Systems*. Vol. 23. 1. sciendo, pp. 21–27. DOI: https://doi.org/10.2478/acss-2018-0003. URL: https://content.sciendo.com/view/journals/acss/23/1/article-p21.xml.